UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TRABAJO FIN DE GRADO

Grado en ingeniería Química

MODELLING THE FREE ENERGY LANDSCAPE OF A MOLECULE

(MODELANDO EL PAISAJE DE ENERGÍA LIBRE DE UNA MOLÉCULA)

Memoria y Anexos

Autor: Rafael León Carrasco
Director: Samir Kanaan Izquierdo

Convocatoria: Junio 2019

Resumen

En este trabajo se tratará de generar distintas conformaciones del ARN mediante redes neuronales a partir de un conjunto de entrenamiento para acelerar el estudio de estas moléculas. También se detallarán los procedimientos para depurar los errores en los datos de entrada, cómo se ha escogido el tipo de red neuronal, su arquitectura y un método para determinar aproximadamente si una conformación, ya sea de las muestras del conjunto de entrenamiento o las generadas por la red neuronal; son de alta energía o no. Esto es útil para saber cuándo una conformación determinada está a punto de cambiar de estado y estudiar el comportamiento de estas moléculas sin simulaciones que pueden durar semanas.

Agradecimientos

Le doy las gracias a Samir Kanaan, el director del proyecto, por tener la paciencia de contestar amablemente a mis innumerables preguntas y revisiones sobre el tema.

También le doy las gracias a Sandro Wrzalek por compartir su trabajo de máster en el cual está basado este proyecto, además de por sus sugerencias para la representación gráfica de las moléculas y por proporcionar los datos para poder entrenar la red neuronal.

Índice

1 Introducción

Como se comenta en la tesis de máster de Sandro Wrzalek: *Mg^{2+} dependent folding behavior of RNA* [1] en la cual se basa éste trabajo, la hipótesis mundial del ARN o ácido ribonucleico propone que la vida surge de la actividad inherente a las moléculas de ARN. Además, juegan un papel importante en las síntesis de proteínas y su transporte; en la regulación y replicación de cromosomas, en la edición de ARN y en la catálisis de ribosomas. Por lo tanto, es vital comprender más en profundidad el comportamiento de éste tipo de moléculas. Si se pudiera comprender el proceso de plegamiento del ARN, se podría utilizar como transporte de moléculas. El ARN se plegaría sobre la molécula y las transportaría mediante un virus hasta el objetivo, donde volvería a desplegarse. Una posible opción para controlar el plegamiento de estas moléculas sería mediante la influencia de iones debido a la alta carga negativa de las mismas. Desafortunadamente, es muy complicado predecir el comportamiento de este tipo de molécula. Se necesitan efectuar pruebas de muy larga duración (semanas) para estudiar cómo se pliega el ARN en distintas condiciones iónicas.

En este proyecto se utilizarán redes neuronales. Estas redes son un tipo de algoritmo, vagamente basado en el funcionamiento del cerebro, diseñado para reconocer patrones. Son extremadamente útiles para clasificar y agrupar datos según las similaridades detectadas por la misma. [9]
Se utilizará un tipo concreto de red neuronal llamado GAN (*Generative Adversary Networks*) o redes generativas antagónicas, donde dos sistemas interactúan entre sí para crear algo completamente nuevo.

El objetivo de este trabajo es crear una red neuronal que permita simular nuevas conformaciones a partir de un conjunto de entrenamiento de una molécula determinada de ARN influenciada por algún tipo de ion como Mg^{2+} o Na^{+}. Este problema se tratará como un problema de aprendizaje automático no supervisado, es decir, los ejemplos suministrados no cuentan con una clase o no están clasificados previamente de ninguna forma, como, por ejemplo, si se encuentra en un estado de alta energía libre o no. Debido a esto no se puede diseñar una red neuronal que clasifique estados de energía libre, pero sí puede crear nuevos ejemplos a partir de los suministrados.

1.1 Objetivos

1. Diseñar un sistema que permita generar nuevas conformaciones a partir de los datos suministrados. Esto se divide en dos partes principales:
    i. Diseñar un método para identificar los datos de entrada erróneos que puedan suponer un problema para el rendimiento de la red.
    ii. Diseñar una red neuronal capaz de generar nuevas conformaciones a partir del punto "i".
2. Como objetivo extra, un sistema que permita (cualitativamente) identificar qué conformaciones son candidatas a ser de alta energía, ya sean de muestras reales o generadas por la red.

1.2 Estructura de la memoria

Los capítulos restantes de esta memoria están estructurados de la siguiente forma:

Descripción y preparación de los datos: en este apartado se describirá el formato de los datos que se usarán para entrenar la red neuronal, además de cómo depurar los posibles errores que contienen.

Creación de la red neuronal: Aquí se describirá el diseño del sistema de generación de conformaciones, así como la arquitectura de la red y cómo se llevará a cabo.

Análisis de resultados: Se evaluarán lo resultados obtenidos y cómo se puede modificar la red para mejorar la calidad de las conformaciones de salida.

Clasificación de estados de alta energía: Como parte opcional del trabajo, se describirá un método para clasificar muestras reales o artificiales como posibles candidatas a ser de alta energía libre.

1.3 Requerimientos previos

Como requerimientos previos para leer este trabajo, se recomienda una comprensión básica del lenguaje de programación Python [2] y con la librería Numpy [8]. También se requiere estar algo familiarizado con redes neuronales y con el framework utilizado: Keras [3].

2 Descripción y preparación de los datos

En este capítulo se analizarán los datos o ejemplos de entrenamiento para detectar posibles errores en los ejemplos con el fin de depurarlos y así poder entrenar la red neuronal con algo más de seguridad.

Además, se analizará un patrón en los datos que puede ser útil para el entrenamiento de la red más adelante.

2.1 Descripción de los ejemplos de entrenamiento

Antes de empezar a construir la red neuronal, es necesario entender los datos que se han suministrado para el entrenamiento, ya sea su formato, clases, etc.

Estos datos han sido suministrados por Sandro Wrzalek y forman parte de su investigación sobre las formas de plegamiento del ARN. Hay varios archivos disponibles para trabajar: *CCC_Mg_aligned.npy, CUC_Mg_aligned.npy, CCC_Na_aligned.npy y CUC_Na_aligned.npy*. Una diferencia entre los archivos está en el ión que se usa para influir en las conformaciones disponibles en cada uno, Mg (Magnesio) o Na (Sodio). También hay dos moléculas disponibles, CCC y CUC; diminutivos del conjunto de bases por las que está formado este ARN, en concreto "AAAAC C CUUUU" y "AAAAC U CUUUU" respectivamente.

En este caso se analizarán los ejemplos suministrados por el archivo *CUC_Mg_aligned.npy*. Este archivo consta de 420040 ejemplos. Cada ejemplo representa una conformación del ARN en la variante CUC influenciado por iones de magnesio. Los ejemplos son "frames" o fotogramas tomados del ARN cada pocos microsegundos con la finalidad de poder captar los estados de mayor energía libre que indican un cambio de forma de plegamiento de la molécula, son representaciones 3D de la conformación.

Concretamente, los fotogramas de la molécula están compuestos por 343 átomos, cada uno de los cuales formado por 3 coordenadas: x, y, z. Por tanto, tenemos como conjunto de entrenamiento una matriz 3D compuesta de la siguiente forma:

Nº de ejemplos	**Nº de átomos por ejemplo**	**Nº de coordenadas por átomo**
420040	343	3

Tabla 1. Composición del conjunto de entrenamiento

Con una sencilla multiplicación se puede comprobar que se dispone de $420040 \cdot 343 \cdot 3 = 432221160$ valores (float32) para usar en la red.

Cabe recordar que estos ejemplos no disponen de una clase o clasificación por lo que se tratará como un problema de aprendizaje no supervisado.

Una representación gráfica de un ejemplo sería la siguiente:

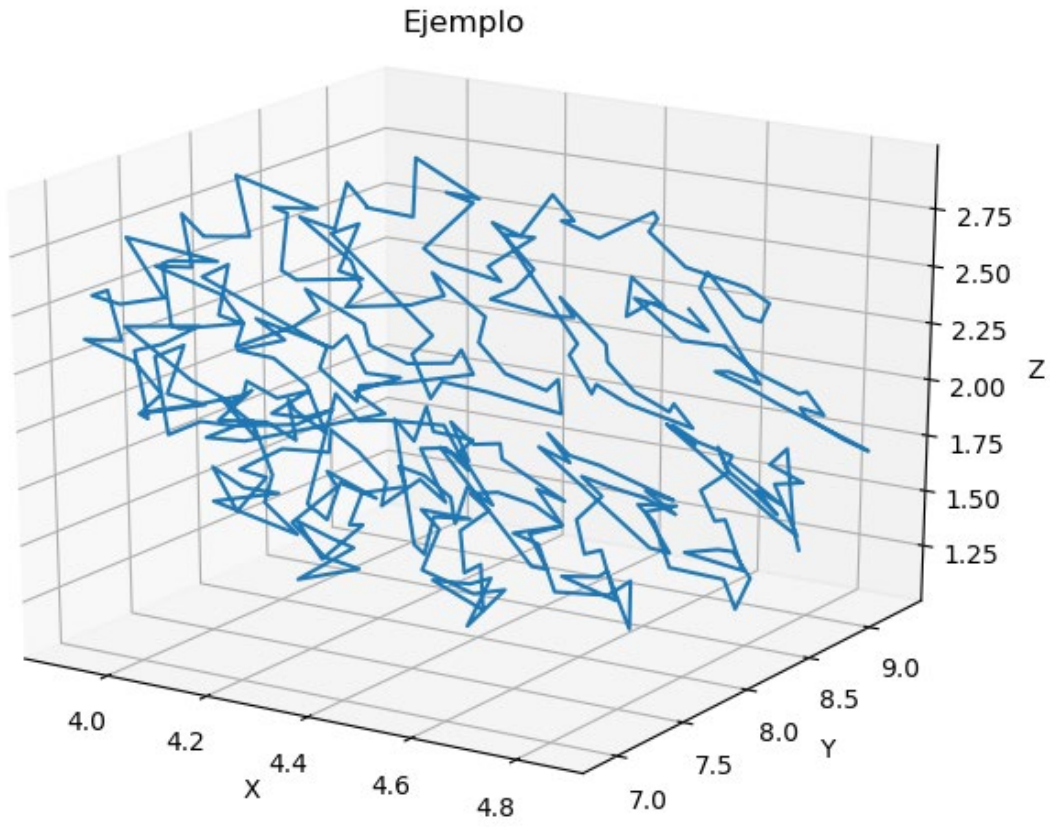


Figura 1. Ejemplo de muestra de entrenamiento

2.2 Depurando lo datos y análisis

Para poder entrenar la red neuronal, hacen falta datos de entrenamiento, pero, ¿qué pasaría si estos datos contuvieran valores erróneos? ¿Qué puede pasar si algunos valores son mucho más grandes o pequeños de lo que se considera "aceptable" en este tipo de problema? Si esto sucede en un conjunto de entrenamiento muy grande, como es este caso, identificar manualmente este tipo de errores es virtualmente imposible, lo que puede provocar que la red neuronal entrenada con estos datos de resultados de mucha menor calidad de la deseada.

Este apartado se centrará en resolver este problema, se identificarán estos errores y se eliminarán para un correcto entrenamiento de la red.

2.2.1 Identificando problemas

¿Cómo se podrían cribar las muestras si se sabe que algunos valores o coordenadas podrían ser mucho mayores o menores de lo normal? Una posible solución a este problema sería calcular (de una manera un poco especial como se verá más adelante) la distancia entre átomo y átomo de cada uno de los ejemplos y después sumarlas, es decir, por cada muestra del conjunto de entrenamiento; al componerse de 343 átomos, se calcularían 342 distancias entre los mismos y después se sumarían. Entonces, si alguna coordenada es defectuosa en cualquier átomo, haría que la distancia calculada en este ejemplo con respecto al anterior y el posterior (si lo hay) se disparara; lo que permitiría identificar una anomalía. Además, se calculará la desviación estándar de las distancias entre átomos de cada muestra para ayudar en este caso.

La ventaja de este método es que es independiente del rango de valores en los que se mueve el ejemplo, es decir, según qué muestras los rangos de coordenadas x,y,z pueden ser bastante diferentes a otras. Por ejemplo, si las coordenadas de la muestra 100000 se sitúan en el siguiente rango:

X	**Y**	**Z**
4.0-4.8	7.0-9.0	1.25-2.75

Tabla 2. Rango de coordenadas del ejemplo 100000 de CUC_Mg

El rango de coordenadas de la muestra 200000 es el siguiente:

X	**Y**	**Z**
7.0-9.0	2.5-5.0	1.4-3.0

Tabla 3. Rango de coordenadas del ejemplo 200000 de CUC_Mg

Por eso es más práctico usar el método de las distancias antes que simplemente encontrar valores discordantes en cada ejemplo teniendo que tener en cuenta el eje y el rango de coordenadas.

La implementación del cálculo de las distancias en Python es el siguiente:

```
data = np.load('C:\Simulation\CUC_Mg_aligned.npy', encoding='bytes')
distancesPath = 'C:\Simulation\CUC_Mg_aligned_distances.npy'
loadDistances = True
```

Código 2.1

Primero se cargan los datos de entrenamiento (*data*) mediante numpy (*np*) y el destino para guardar las distancias una vez calculadas o para cargarlas si ya han sido previamente generadas. Esto se controla cambiando manualmente el valor de loadDistances a *True* si ya han sido creadas o *False* si se quieren generar estos valores.

```
distance = []

if(not loadDistances):
    for j, example in enumerate(data):
        dist = [0,0]#EL primero la distancia, el segundo la desviación
estándard
        forstd = []
        for i in range(1, len(example)):#Empieza en uno
            p1 = np.array([example[i-1][0], example[i-1][1], example[i-
1][2]])
            p2 = np.array([example[i][0], example[i][1], example[i][2]])

            squared_dist = np.sum(p1**2 + p2**2)
            forstd.append(squared_dist)
            dist[0] += np.sqrt(squared_dist)

        dist[1] = np.std(forstd)
        distance.append(dist)
    distance = np.asarray(distance)
    np.save(distancesPath, distance)
else:
    distance = np.load(distancesPath)
```

Código 2.2

En cada elemento de la variable *distance* se guardarán pequeñas *arrays* de 2 valores de longitud donde el primero será la distancia total que tiene cada ejemplo entre átomos, y el segundo la desviación estándar. Concretamente, si *loadDistances* es *False* se recorrerá muestra a muestra los datos y, dentro de cada muestra, calculará las distancias que hay entre cada par vecino de átomos y las irá sumando hasta tener la distancia total de la molécula. Además, calculará la desviación estándar de las distancias entre átomos para cada muestra con *np.std()* una vez haya calculado el ejemplo entero.

Por último, cabe comentar que la distancia cuadrática se calcula con `np.sum(p1**2 + p2**2)` en vez de algo como `np.sum((p1-p2)**2)` porque, aunque no sea realmente la distancia lo que se calcula; permite dibujar una gráfica mucho más clara al acentuar las diferencias que ya son grandes de por sí. Se puede ver una explicación más detallada en el *anexo 1*.

Si ahora se imprimen las medias y máximos de los valores obtenidos mediante la siguiente función aplicada a la variable *distance*:

```
def printMeansAndCo(noInf_dist):
    print("Valor medio máximo:",np.amax(noInf_dist[:,0]))
    print("Posición del valor medio máximo",np.where(noInf_dist ==
np.amax(noInf_dist[:,0])))
    print("Valor medio mínimo:",np.amin(noInf_dist[:,0]))
    print("Desviación estándard máxima:",np.amax(noInf_dist[:,1]))
    print("Posición del std máximo",np.where(noInf_dist ==
np.amax(noInf_dist[:,1])))
```

Código 2.3

Se obtiene:

```
Valor medio máximo: inf
Posición del valor medio máximo (array([  1003,   1004,   1005,   1006, …
], dtype=int64), array([0, 0, 0,… ], dtype=int64))
Valor medio mínimo: 33.51812160201371
Desviación estándard máxima: nan
Posición del std máximo (array([], dtype=int64), array([], dtype=int64))
```

Salida 2.1

Se observa que el valor medio máximo es infinito, por lo que el primer paso será eliminar esos valores.

```
data = data[distance[:,0] < inf]
noInf_dist = distance[distance[:,0] < inf]
```

Código 2.4

De esta forma se han eliminado todos los valores infinitos en *data* según los valores de *distance*. También se han eliminado de la propia *distance* y guardado por conveniencia en una nueva variable llamada *noInf_dist*.

Si ahora se observan las medias y máximas como anteriormente para *noInf_*dist se obtiene:

```
Valor medio máximo: 1522691.4466301128
Posición del valor medio máximo (array([122604], dtype=int64), array([0],
dtype=int64))
Valor medio mínimo: 33.51812160201371
Desviación estándard máxima: 18552252416.0
Posición del std máximo (array([125653], dtype=int64), array([1],
dtype=int64))
```

Salida 2.2

Ya no se observan valores infinitos, entonces se graficarán las distancias y la desviación estándar a lo largo de los ejemplos para comprobar si se mantienen más o menos constantes:

```
def drawGraphs(chunk_size):
    def chunks(a, chunk_size):
        counter = 0
        output = []
        if(chunk_size > 0 and chunk_size <= len(a)):
            while counter+chunk_size <= len(a):
                output.append(a[counter:counter+chunk_size])
                counter += chunk_size
            if(counter < len(a)):
                output.append(a[counter:len(a)])

        return output

    chunkos = chunks(noInf_dist, chunk_size)
    means = []
    for item in chunkos:
        temp = [0,0]
        temp[0] = np.mean(item[:,0])
        temp[1] = np.std(item[:,1])
        means.append(temp)
    means = np.asarray(means)

    plt.figure(1)
    plt.plot(list(range(0,len(means))), means[:,0])
    plt.xlabel('Ejemplos x'+str(chunk_size))
    plt.ylabel('Media por cada '+str(chunk_size))
    plt.title("Medias por cada "+str(chunk_size))
    plt.figure(2)
    plt.plot(list(range(0,len(means))), means[:,1])
    plt.xlabel('Ejemplos x'+str(chunk_size))
    plt.ylabel('Std media por cada '+str(chunk_size))
    plt.title("Std por cada "+str(chunk_size))
    plt.show()

    print("Valor de data máximo:",np.amax(data))
    print("Valor de data mínimo:",np.amin(data))
```

Código 2.5

En esta parte se dibuja haciendo una media de las distancias y desviación estándar cada 50 ejemplos (*chunk_size* = 50) para hacer las gráficas un poco más legibles. El resultado es el siguiente:

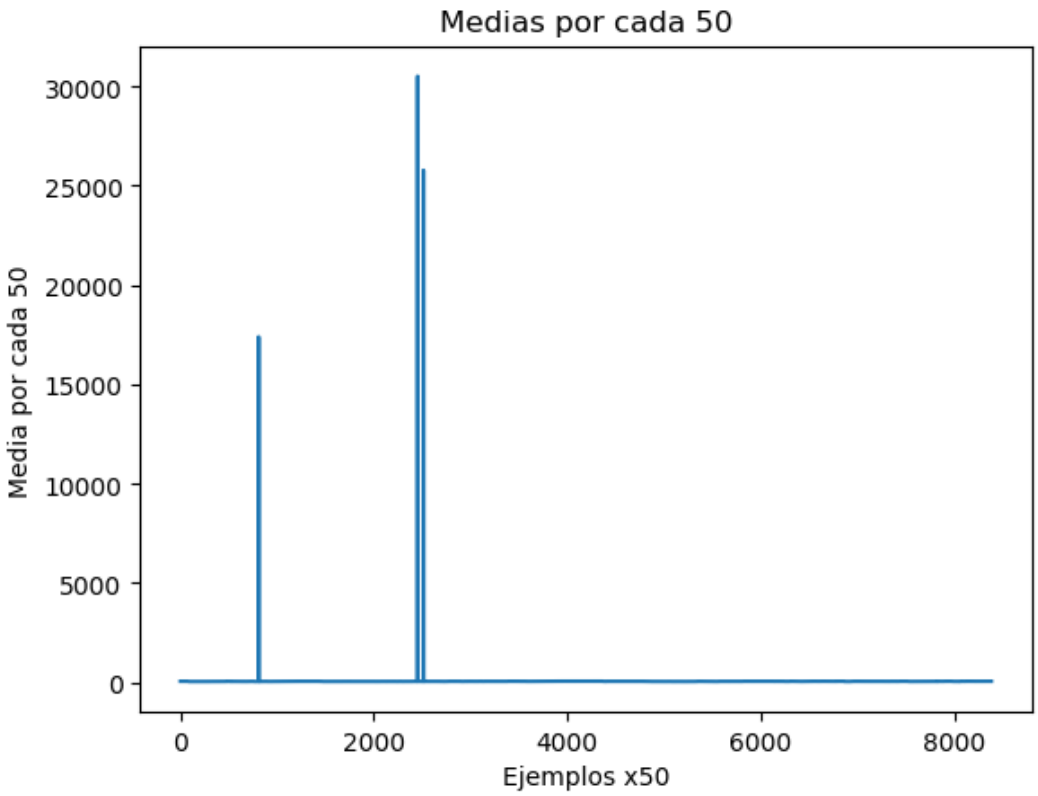


Figura 2. Medias de distancia sin filtrar

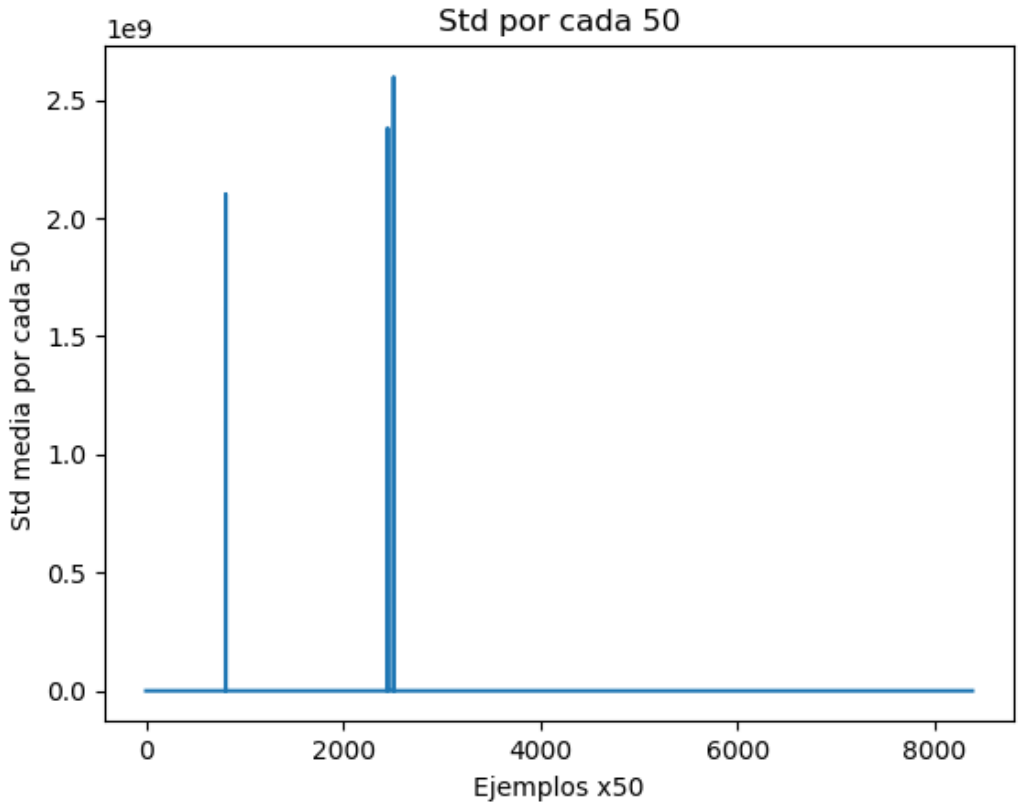


Figura 3. Medias de desviación estándar sin filtrar

Se puede concluir que hay algunos valores tan altos en ambas gráficas, que hacen que el resto se vea plano y, por lo tanto, son seguramente valores erróneos. Ahora, usando, por ejemplo; la gráfica de la *Figura 3* (desviación estándar) se filtrarán los valores eliminando todos los superiores a, por ejemplo, 1000; así se podrán volver a dibujar comprobando si esta vez los valores son correctos.

```
stdFilterValue = 1000
data = data[noInf_dist[:,1] < stdFilterValue]
noInf_dist = noInf_dist[noInf_dist[:,1] < stdFilterValue]
```

Código 2.6

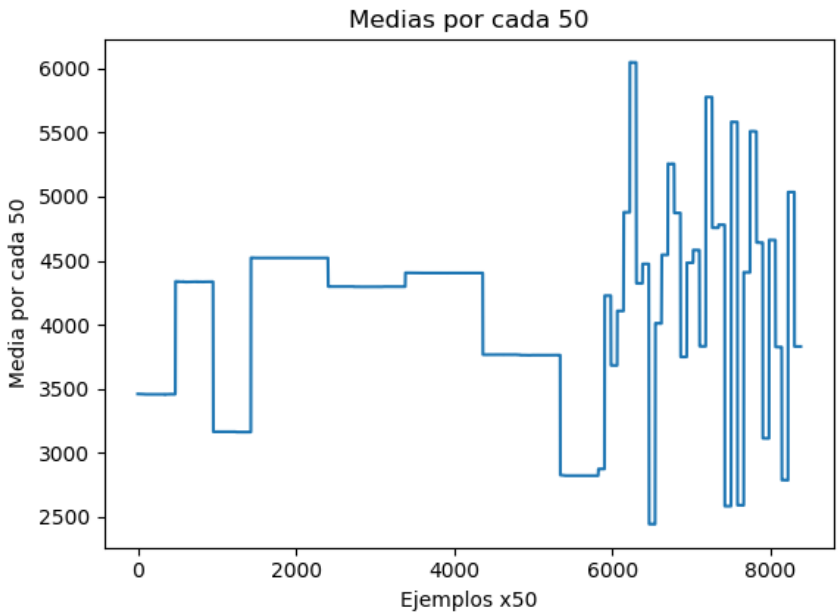


Figura 4. Medias de distancia filtradas

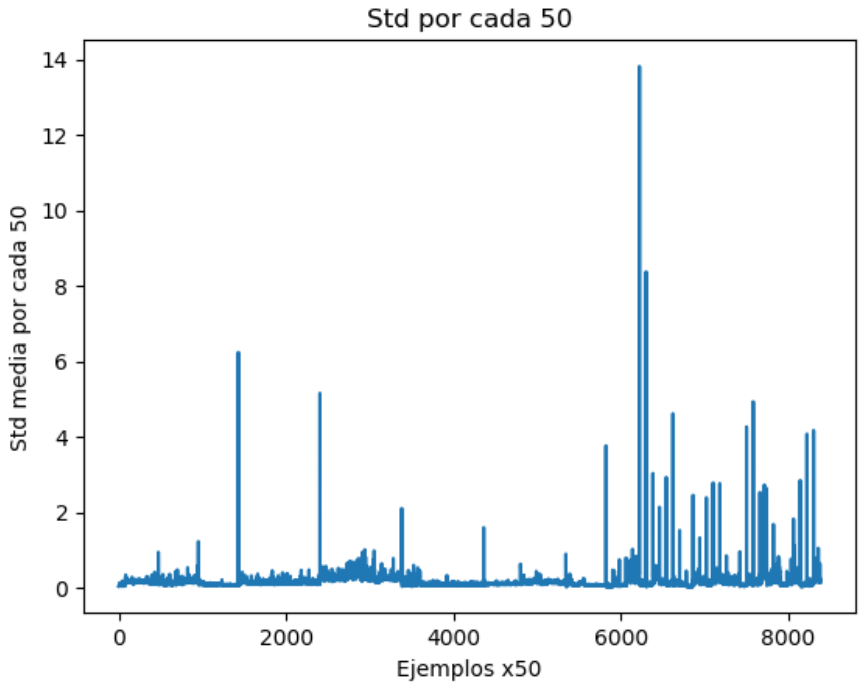


Figura 5. Medias de desviación estándar filtradas

```
Valor medio máximo: 6048.997517585754
Posición del valor medio máximo (array([311181], dtype=int64), array([0],
dtype=int64))
Valor medio mínimo: 2439.3262157440186
Desviación estándard máxima: 38.65486145019531
Posición del std máximo (array([311248], dtype=int64), array([1],
dtype=int64))
```

Salida 2.3

En la *figura 4* y en la *figura* 5 ya no se observan valores extremadamente altos o bajos y, en el caso de que los hubieran; no habría más que repetir el procedimiento disminuyendo el valor de la variable *stdFilterValue* hasta que las gráficas parezcan correctas. Además, se puede comprobar en el último *output* como los valores máximos y mínimos en distancias y en desviación estándar son mucho más razonables que en los anteriores ejemplos.

2.2.2 Un patrón en los datos

En la *figura 4* se puede deducir una especie de patrón, donde se ve que las distancias medias calculadas en la primera mitad de ejemplos, hasta el valor 6000 en el gráfico, mantienen constantes sus valores en intervalos más o menos largos mientras que después los intervalos se vuelven muchísimo más cortos entre sí, aunque de idéntica longitud.

El siguiente paso será extraer estos datos del patrón que pueden ser útiles más adelante. En concreto, se buscarán los índices en el conjunto de entrenamiento donde se producen estos saltos en los valores.

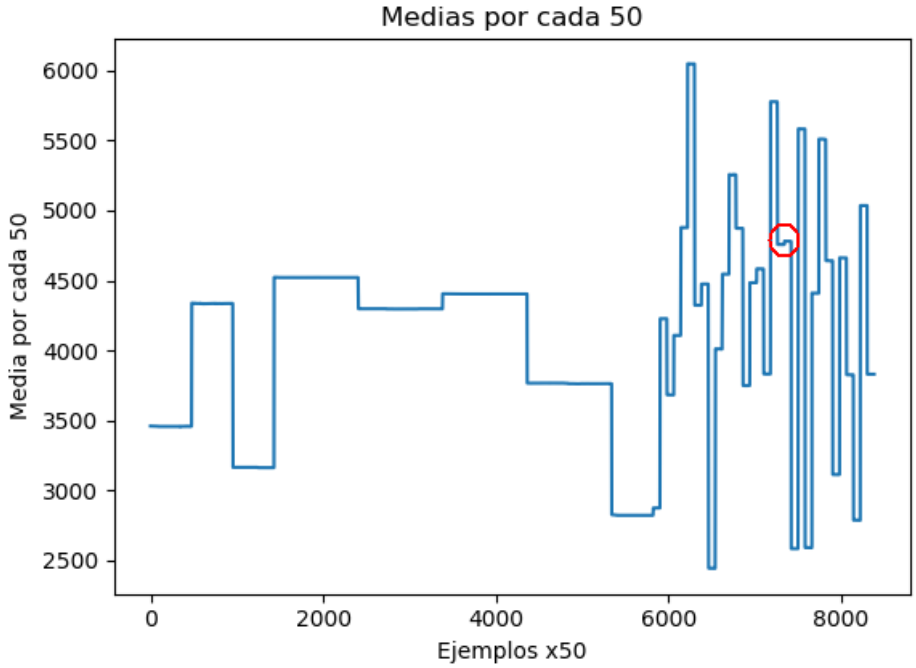


Figura 6. Medias de distancia filtradas

Una posible solución a este problema para obtener los saltos entre valores, sería buscar manualmente en el gráfico cuál es el valor del menor salto que se produce y entonces crear una lista con todos los índices de estos saltos según esto.

Buscando el valor del salto mínimo en el gráfico, como se ve en la *figura* 6, se obtiene un valor próximo a 30 (en el eje Y). Ahora se escoge un valor ligeramente inferior a 30 como 15 para evitar posibles problemas de detección de estos saltos y se utiliza en el siguiente código:

```
minimumJumpLength = 15
jumpIndexes = []
for i in range(1, len(noInf_dist)):
    if(abs(noInf_dist[i,0] - noInf_dist[i-1,0]) > minimumJumpLength):
        jumpIndexes.append(i)

jumpIndexes = [0] + jumpIndexes + [len(noInf_dist)]
```

Código 2.7

Se compara en cada par adyacente de ejemplos las distancias filtradas de errores y, si el valor es superior a 15 (*minimumJumpLength*), se clasifica como salto y se guarda el índice en la variable *jumpIndexes*. Entonces se añade el índice 0 al principio de *jumpIndexes* y el índice de la longitud del

conjunto de entrenamiento (o la longitud de *noInf_Dist,* que es idéntica) para que se pueda utilizar más adelante si se quiere escoger un rango de valores determinado.

Ahora, *jumpIndexes* tiene un aspecto parecido a esto:

[0, 23799, 47687, 71564, 120327, 169173, 218174, 267175, 291176, 295177, 299178, 303179, 307180, 311181, 315182, 319183, 323184, 327185, 331186, 335187, 339188, 343189, 347190, 351191, 355192, 359193, 363194, 367195, 371196, 375197, 379198, 383199, 387200, 391201, 395202, 399203, 403204, 407205, 411206, 415207, 419208]

Salida 2.4

De hecho, se puede comprobar que *jumpIndexes* tiene 41 valores, número que coincide con el número de saltos en la *figura 6*, por lo que se puede asegurar que se han detectado los saltos correctamente.

Por último, si se comprueba la diferencia que hay entre valores consecutivos en *jumpIndexes* se obtiene lo siguiente:

[23799, 23888, 23877, 48763, 48846, 49001, 49001, 24001, 4001]

Salida 2.5

Se deduce que en los datos de entrenamiento se han mezclado muchas conformaciones distintas de la misma molécula. Es decir, se han mezclado muchas pruebas de 4001 *frames* en un momento dado de la molécula con otras pruebas más largas de 49001 o 24001. También se pueden ver algunos números al principio que no son redondos, no acaban en “001”. Esto es debido a los ejemplos erróneos que se han eliminado previamente. Para comprobar que, efectivamente, se tratan de distintas pruebas hechas en distintas fases de la molécula; se compararán 2 ejemplos distintos obtenidos de la misma prueba con otros dos de otra distinta.

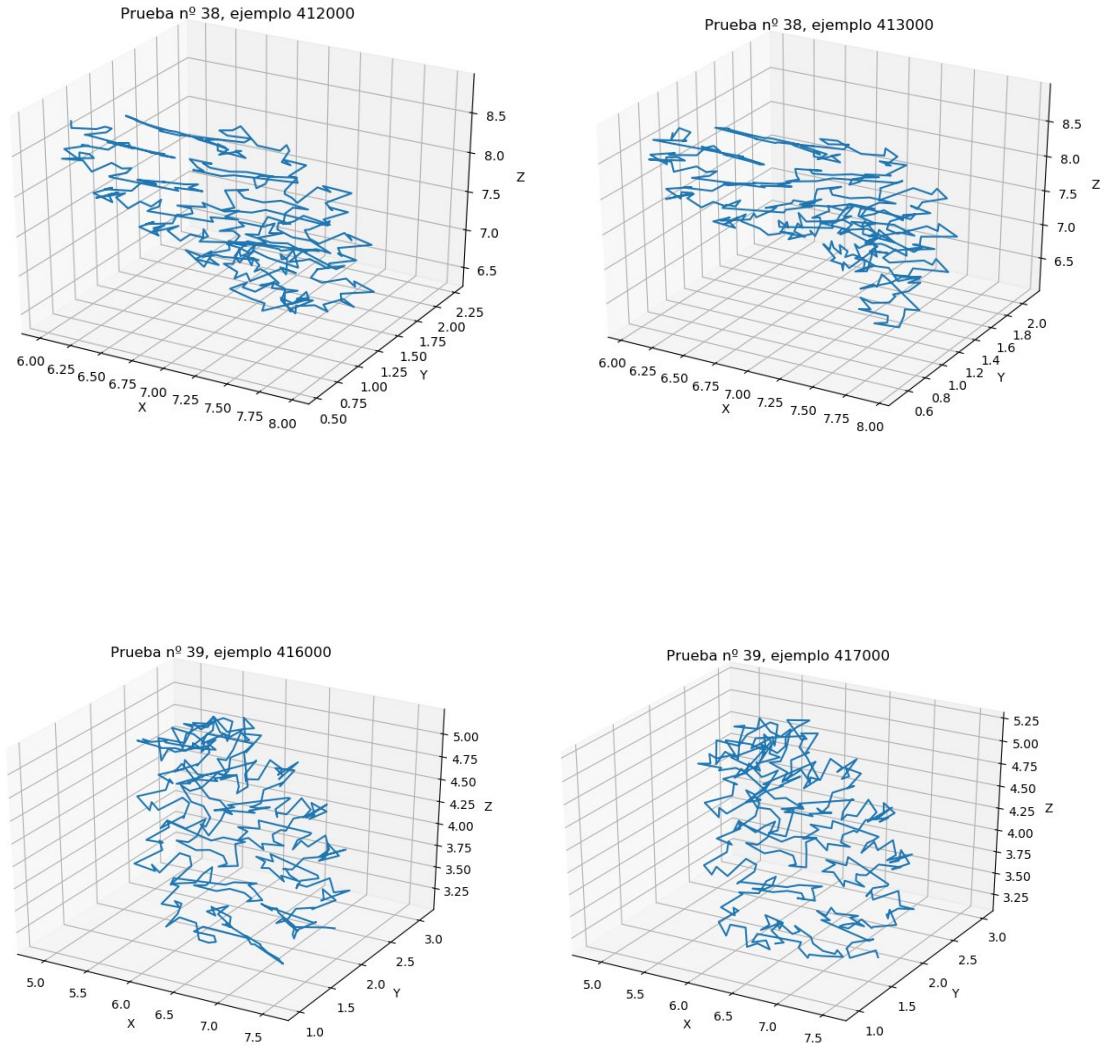


Figura 7. Comparación de distintas pruebas

Se comprueba como comparando dos ejemplos de la prueba 38 (411206-415207) con otros dos de la prueba 39 (415207-419208) la conformación de la molécula es completamente distinta entre pruebas, con lo que se puede decir que los valores de los índices de los saltos (*jumpIndexes*) son correctos.

3 Creación de la red neuronal

En este capítulo se empezará a desarrollar la red neuronal, se explicará el tipo que se ha escogido, así como su arquitectura y de qué forma se utilizará el conjunto de entrenamiento obtenido en el apartado 2. Depurando datos y análisis.

3.1 Tipo de red neuronal

Como se ha comentado anteriormente, el conjunto de entrenamiento no dispone de ningún tipo de clasificación o clases por lo que se trata de un problema de aprendizaje no supervisado.

En este caso, se ha escogido una red *GAN* con *autoencode*r debido a su rapidez de procesamiento en comparación con otro tipo de redes como las GAN convolucionales que son adecuadas para reconocer patrones en imágenes. Una GAN o red generativa antagónica, es un sistema basado en dos redes neuronales que compiten entre sí. Una red generadora intenta generar conformaciones cada vez más realistas para engañar a la otra parte, la red discriminativa, que busca discernir entre las que son reales (del conjunto de entrenamiento) y las falsas de la red generadora.

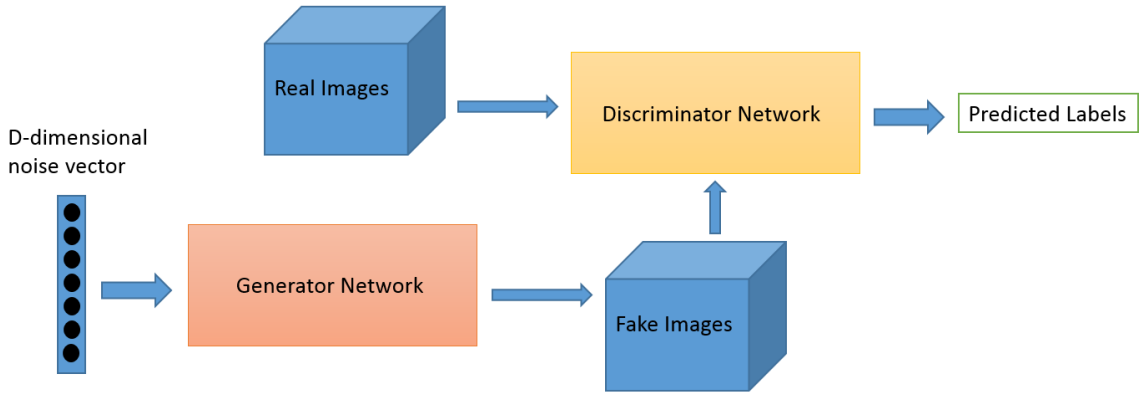


Figura 8. Diagrama de una GAN

Para aclarar un poco más como fuciona una GAN, se utilizará un ejemplo de clasificación de imágenes en la *figura 8* [10]. A partir de un vector completamente al azar de *D* dimensiones, la red generadora es capaz de crear una imagen parecida a una real con el objetivo de engañar al discriminante. Por otro lado, la red discriminativa; la cual es alimentada por las imágenes falsas de la red generadora e imágenes verdaderas, intenta calsificar correctamente las imágenes como reales o no. Al repetir este proceso miles de veces, las dos redes mejoran su capacidad de generar nuevas imágenes y de clasificar

correctamente; lo que permite que la parte generadora sea lo bastante “buena” para crear imágenes realistas a partir de vectores aleatorios.

Un ejemplo de aplicación de las GAN es el siguiente:

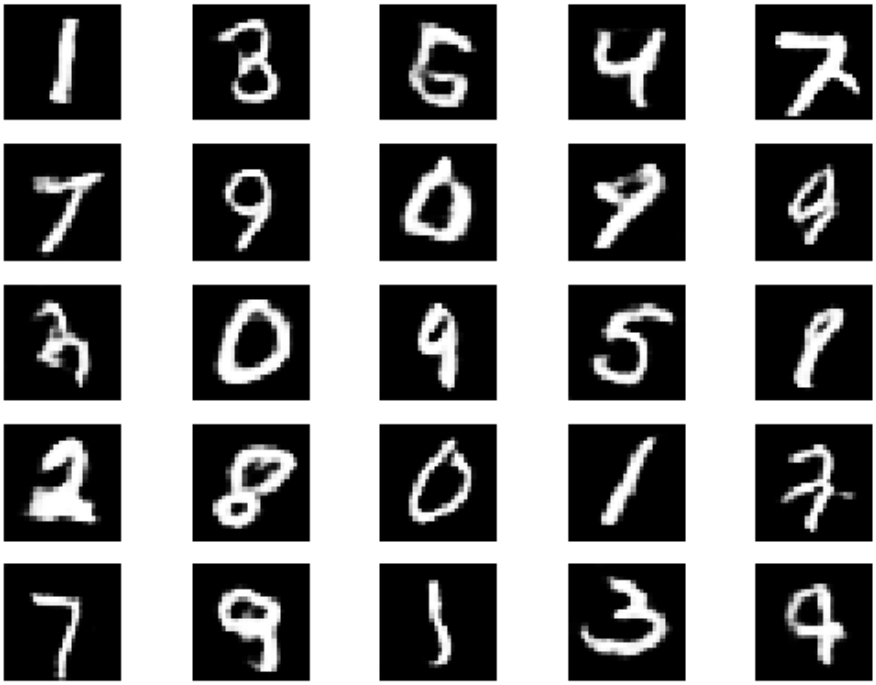


Figura 9. Números generados por una GAN

En la *figura 9* se muestran números generados artificialmente por una GAN cuyo conjunto de entrenamiento son números hechos a mano. Esto muestra que podría ser útil para generar nuevas conformaciones a partir de muestras reales.

Por otro lado, el *autoencoder* es un modo de implementar la red basado en reducir la dimensionalidad de la conformación de entrada, compuesta de 343 átomos con sus respectivas 3 coordenadas, es decir 1029 valores; a, por ejemplo, sólo 300; esta es la parte llamada *encoder*. Después, a partir de esos 300 valores, se intenta representar de nuevo la conformación original de 1029 coordenadas; esto es el *decoder*. Durante este proceso de entrenamiento, la parte del *decoder* crea un modo de generar conformaciones parecidas a las reales a partir de los 300 valores aportados por el *encoder*; entonces, una vez la red ha sido entrenada, el *decoder* debería ser capaz de generar nuevas conformaciones a partir de valores al azar.

Los ejemplos del conjunto de entrenamiento que se utilizará son moléculas en un espacio tridimensional, por lo que una red convolucional 3D podría captar muy bien este tipo de patrones, pero la capacidad de procesamiento requerido para esto descarta esta opción. El código se basa en la implementación de Erik Linder-Norén [4] y el artículo *Adversarial Autoencoders* [5]

3.2 Implementación de la red y arquitectura

La GAN se implementará utilizando Keras.

Primero se definirán las dimensiones de las conformaciones de entrada y la cantidad de dimensiones a las cuales se reducirá para el *encoder*.

```
# Input shape
 self.num_atoms = 343#Son los átomos por muestra
 self.dimensions = 3#Las coordenadas por átomo
 self.sample_shape = (self.num_atoms, self.dimensions)
 self.latent_dim = 343#Las dimensiones a las cuales reducir el input
```

Código 3.1

Se reducirá de 1029 valores a un tercio del total 343 (llamadas dimensiones latentes), debido a que se requiere que las conformaciones generadas sean lo más reales posible, entonces cada átomo se representará por un valor en el *encoder* buscando la mayor precisión posible esperando que tampoco se parezcan demasiado a los ejemplos reales.

3.2.1 Arquitectura de la red

La arquitectura del *encoder* es la siguiente:

```
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
        h = Dropout(0.25)(h)
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
        h = Dropout(0.25)(h)
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
        h = Dropout(0.25)(h)

        #https://arxiv.org/pdf/1511.05644.pdf
```

```
        mu = Dense(self.latent_dim)(h)
        log_var = Dense(self.latent_dim)(h)

        latent_repr = Lambda(lambda p: p[0] +
K.random_normal(K.shape(p[0])) * K.exp(p[1] / 2))([mu, log_var])
```

Código 3.2

Antes de nada, se explicará qué es *LeakyReLU, Dropout y Dense*.

LeakyReLU: Es una función de activación que se suele utilizar en GANs, ayuda a evitar que la red no se quede "atascada" en el entrenamiento en comparación con la función *sigmoid* o *relu*. El valor 0.2 es el valor

Dropout: Es una función de regularización de la red. Si una red no se regulariza, es propensa a producir un sobreajuste u *overfit*; esto es que los resultados son propensos a parecerse demasiado al conjunto de entrenamiento. Con la regularización se evita esto y permite generar conformaciones nuevas más fácilmente. El valor 0.25 quiere decir que se desactivarán al azar el 25% de las neuronas de esa capa en ese ciclo de entrenamiento.

Dense: Es una llamada estándar que define el número de neuronas que habrá en una capa determinada.

El *encoder* se compone de 3 capas de 1024 neuronas cada una, con un input de los 1029 valores de los que se compone cada conformación; aplicando la activación *LeakyReLU* y una regularización *Dropout* a cada paso. Por último, se crean dos valores de salida de la última capa, *mu* y *log_var*; que son la media y la desviación estándar; luego se vuelven a mezclar sumando a la media un valor al azar calculado mediante la fórmula de una distribución gaussiana. La forma de salida es un vector de 343 valores, las dimensiones latentes.

Ésta es la arquitectura del *decoder*:

```
        model.add(Dense(1024, input_dim=self.latent_dim))

        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Dense(1024))

        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Dense(1024))

        model.add(LeakyReLU(alpha=0.2))
```

```
model.add(Dropout(0.25))

model.add(Dense(np.prod(self.sample_shape), activation='tanh'))
model.add(Reshape(self.sample_shape))
```

Código 3.3

La arquitectura central es idéntica al del *encoder*, la diferencia está en los valores de entrada; tienen el tamaño de las dimensiones latentes (343) y también, después de las 3 capas de 1024 neuronas, se reescala de nuevo a 1029 usando una función de activación *tanh.* Por último, se devuelve la forma original a la conformación de 343 átomos con 3 coordenadas cada uno.

La última arquitectura a describir será el *dicriminator*.

```
model = Sequential()

model.add(Dense(512, input_dim=self.latent_dim))

model.add(LeakyReLU(alpha=0.2))
model.add(Dense(256))

model.add(LeakyReLU(alpha=0.2))
model.add(Dense(1, activation="sigmoid"))
model.summary()
```

Código 3.4

Ésta parte es la encargada de decidir si la conformación codificada por el *encoder* es real o falsa. Consta de la capa de input de 512 neuronas, que está conectada a la de 256; ambas con la activación *LeakyReLU*. La siguiente capa consta de una sola neurona con la activación *sigmoid* que da las probabilidades de que la conformación sea real o no.

3.2.2 Hiperparámetros

El siguiente paso es compilar eligiendo los hiperparámetros adecuados. Como optimizador se usará *Adam*, el cual tiene un rendimiento general superior en la mayoría de situaciones [6].

Para el *discriminator* se usará la función de coste *binary_crossentropy*, adecuada para una clasificación binaria (verdadero o falso) como es este caso. La métrica que se utilizará para medir si está teniendo un buen rendimiento será *accuracy*, es decir, medirá la precisión que se ha tenido al catalogar cada muestra como falsa o verdadera.

```
self.discriminator.compile(loss='binary_crossentropy',
            optimizer=self.optimizer,
            metrics=['accuracy'])
```

Código 3.5

El *autoencoder* se compilará de la siguiente manera:

```
        self.encoder = self.build_encoder()
        self.decoder = self.build_decoder()

        img = Input(shape=self.sample_shape)

        encoded_repr = self.encoder(img)
        reconstructed_img = self.decoder(encoded_repr)

        self.discriminator.trainable = False

        validity = self.discriminator(encoded_repr)

        self.adversarial_autoencoder = Model(img, [reconstructed_img,
validity])
        self.adversarial_autoencoder.compile(loss=['mse',
'binary_crossentropy'],
            loss_weights=[0.999, 0.001],
            optimizer=self.optimizer)
```

Código 3.6

Se construirá un modelo donde el *input* es codificado por el *encoder* y después el resultado es, por un lado, clasificado en el *discriminator* y por otro es decodificado en el *decoder*. Antes de la compilación se especifica que en este modelo no se debe entrenar el *dicriminator*, que será entrenado a parte.

En la parte de la compilación se establece el mismo *optimizer Adam*; como funciones de coste se establecen *mse* (*medium squared error*) para medir la precisión de la conformación reconstruida en relación a las muestras reales, y *binary crosentrophy* para el *discriminator*. Además, la calidad del resultado de la decodificación tendrá mucho más peso en el entrenamiento (0.999) que el *discriminator* (0.001).

3.2.3 Normalización de los datos

Para que el rendimiento de la red sea mayor, es necesario normalizar los datos. En este caso se han normalizado en un rango de -1 a 1, usando los *jumpIndexes* que marcaban las distintas muestras o *runs* en los datos. Como diferentes conformaciones en diferentes *runs* pueden estar en un rango de valores de x,y,z muy diferentes entre sí. Es necesario que las normalizaciones se hagan sólo entre muestras de la misma "hornada" para evitar que se formen conformaciones normalizadas muy alejadas entre sí en el espacio. Por ejemplo, una conformación perteneciente al primer *run* donde los rangos de x,y,z son; respectivamente (7-8), (9-10) y (7-9), una vez normalizado puede tener valores entre 0.6 y 0.7; en cambio, una conformación que pertenezca al *run* 30 cuyos rangos sin normalizar para x,y,z son (2-3), (3,4) y (1-3), normalizado podría tener unos valores entre -0.5 y -0.7. Si los valores se estandarizan usando *jumpIndexes*, todos los *runs* tendrán un rango de valores normalizado mucho más abierto y similar entre sí.

Se implementa de la siguiente manera, donde *X_train* es el conjunto de entrenamiento y *jumpIndexes* los índices donde cambia el *run*:

```
        scaler = MinMaxScaler(feature_range=(-1, 1))
        for j in range(0,len(jumpIndexes)-1):
            minMaxItem = []
            start = jumpIndexes[j]
            end = jumpIndexes[j + 1]
            for i in range(X_train.shape[2]):
                subItem = []
                subItem.append(np.amin(X_train[start:end, :, i]))
                subItem.append(np.amax(X_train[start:end, :, i]))
                minMaxItem.append(subItem)
                X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])
```

Código 3.7

3.2.4 Entrenamiento

La parte más importante de la fase de entrenamiento es ésta:

```
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            conf = X_train[idx]

            latent_fake = self.encoder.predict(conf)

            latent_real = np.random.normal(size=(batch_size,
self.latent_dim))

            d_loss_real = self.discriminator.train_on_batch(latent_real,
valid)
            d_loss_fake = self.discriminator.train_on_batch(latent_fake,
fake)

            g_loss = self.adversarial_autoencoder.train_on_batch(conf,
[conf, valid])
```

Código 3.8

Este código se ejecutará X veces (depende de los *epochs*) escogiendo conformaciones reales al azar que se guardan en *conf*. A continuación, las conformaciones se codifican en el *encoder* que se guarda en la variable *latent_fake* y se genera ruido al azar para *latent_real*. El siguiente paso es entrenar el *discriminator* para que clasifique los datos de *latent_real* como verdaderos y de *latent_fake* como falsos. Por último, se entrena el *autoencoder* para que genere nuevas conformaciones lo más parecidas posibles a las reales haciendo a la vez que el *discriminator* clasifique las conformaciones de verdad como falsas. Como aclaración, el *discriminator* está siendo entrenado al revés, está intentando clasificar las imágenes falsas como válidas y las reales como falsas. Esto también da resultados correctos porque el *decoder*, una vez acabado el entrenamiento, generará conformaciones lo suficientemente buenas, sean reales o falsas. En este caso en particular, si el *discriminator* es entrenado para clasificar las muestras reales como reales al igual que el *autoencoder*, el sistema ni siquiera converge.

Esto generará conformaciones ya normalizadas de -1 a 1, por lo que más adelante convendría devolverlas a su tamaño original.

La pérdida en esta GAN es la siguiente:

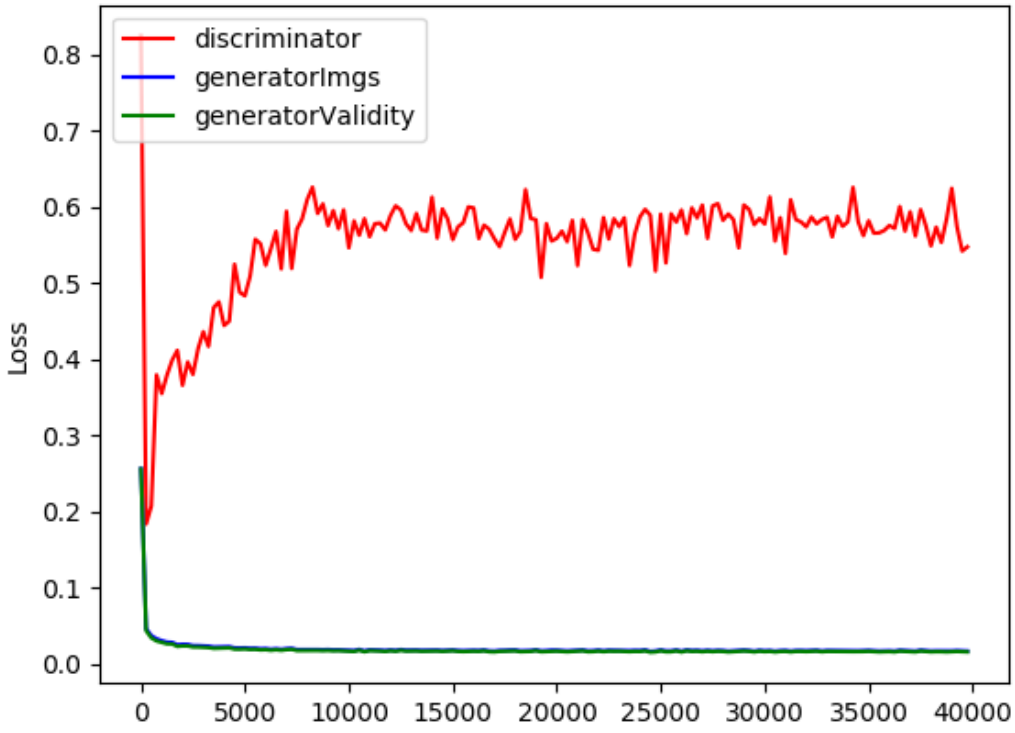


Figura 10. Pérdida en GAN Autoencoder

Esta GAN ha sido entrenada durante 40000 *epochs* en *batchs* de 512, es decir, en cada epoch se han elegido 512 muestras al azar del conjunto de entrenamiento y se observa como el *generator,* es decir, el *autoencoder,* no puede reducir más su función de coste por lo que las conformaciones generadas no verán una mejora de calidad aun siendo entrenada durante más tiempo.

4 Análisis de Resultados

En este apartado se analizarán las muestras generadas por la red neuronal y se propondrán mejoras para la misma.

4.1 Evaluación de resultados preliminares

Para evaluar la calidad de los resultados, para cada muestra generada por la GAN, se comparará con los *runs* normalizados de -1 a 1 del conjunto de entrenamiento mediante *mse (medium squared error)*, y el grupo con el cual el *mse* sea mínimo se establecerá como al que pertenece. Entonces se procederá a "desnormalizar" o volver al tamaño original a la conformación generada según al *run* al que pertenezca. Esto se hará de la siguiente manera:

```
scalers = []

for j in range(0,len(jumpIndexes)-1):

    start = jumpIndexes[j]
    end = jumpIndexes[j + 1]
    for i in range(X_train.shape[2]):
        scaler = MinMaxScaler(feature_range=(-1, 1))
        X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])
        scalers.append(scaler)
```

Código 4.1

En esta parte simplemente, para cada *run* marcado por los índices en *jumpIndexes,* normaliza los valores de -1 a 1 y guarda el escalado de ese *run* en concreto en la variable *scalers* para, más adelante, poder devolver el tamaño original a las muestras generadas que puedan pertenecer al mismo.

El siguiente paso para poder comparar muestras generadas con las reales de una forma ágil es calcular la media de valores de cada átomo para cada *run*. De esta forma cada conformación generada, si sólo hay 40 *runs* diferentes, deberá compararse con éstos 40 en vez de con todos los ejemplos del conjunto de entrenamiento (alrededor de 400000) lo que impactaría seriamente en la velocidad del procesado.

Se implementaría así:

```
X_train_mean = []
for j in range(0,len(jumpIndexes)-1):
    tempTrain = X_train[jumpIndexes[j]:jumpIndexes[j+1]]
```

```
    X_train_mean.append(np.mean(tempTrain,axis=0))
X_train_mean = np.asarray(X_train_mean)
```

Código 4.2

Ahora se deben comparar las muestras generadas con las medias calculadas en *X_train_mean* y clasificarlas en un *run* o en otro. Después se contarán el número de muestras clasificadas en cada *run* para determinar si cada tipo de conformación está correctamente representada.

```
gen_samples =
np.load('C:\\Simulation\\savedModels\\%s\\CUC_Na_aligned_GAN_gen_all.npy'
% folder, encoding='bytes')

mse = []
for i, gen in enumerate(gen_samples):

    mseItem = []
    mseMean = []
    for item in X_train_mean:
        mseItem.append(((gen - item)**2).mean())
    for j in range(0,len(jumpIndexes)-1):
        mseMean.append(np.mean(mseItem[j:j+1]))
    mse.append(mseMean)
classif = []
for item in mse:
    classif.append(int(np.where(item==np.amin(item))[0]))
normDict = {i:classif.count(i) for i in classif}
```

Código 4.3

Primero se cargan las muestras generadas (1000) en *gen_samples*. El siguiente paso es calcular la diferencia media de cada elemento generado mediante *mse* con las medias de cada *run* en *X_train_mean*. El *mse* se calcula según: ((gen - item)**2).mean(); esto calcula la diferencia de cada coordenada entre la conformación generada y la media y eleva el resultado al cuadrado, posteriormente se hace una media con todos los valores para obtener la diferencia con ese *run* en concreto. Se acaba obteniendo la variable *mse* que contiene el *medium squared error* de cada una de las 1000 muestras con cada uno de los 40 *runs* por lo que queda una matriz con una forma de 1000x40. Entonces, se crea una variable llamado *classif* que guarda para cada conformación artificial la posición del *run* con el cual se lleva menos diferencia para así poder devolverlas a su tamaño real más adelante. Por último, se construye el diccionario *normDict* que cuenta el número de valores repetidos que hay

en *classif*, esto servirá para saber si un tipo de conformación en concreto está mucho más repetida que otra o si, directamente, no existe.

Un ejemplo de los valores que devuelve la variable *mse* para la última de las muestras generadas (la número 1000) es el siguiente:

[0.097125076, 0.124266855, 0.13048767, 0.16155824, 0.15474577, 0.050719265, 0.08042268, 0.10133482, 0.10107754, 0.10065949, 0.09553194, 0.11153388, 0.10224382, 0.091563635, 0.08227976, 0.06329463, 0.105368465, 0.08663877, 0.102130294, 0.09410698, 0.10243912, 0.14294425, 0.13326539, 0.08148062, 0.1069361, 0.14262307, 0.10906373, 0.08582456, 0.1270889, 0.09658725, 0.08905788, 0.10521568, 0.09712196, 0.09006366, 0.094596155, 0.112920925, 0.097922705, 0.07301191, 0.100530796, 0.0966191]

Salida 4.1

Esto muestra que el *run* con el que se lleva menos diferencia es el que está en sexta posición (0.050719265). La diferencia en relación a los demás es suficientemente grande como para estar seguros de que la muestra está correctamente clasificada.

Sólo falta restaurar el tamaño original de las muestras artificiales usando la variable *scalers* generada anteriormente:

```
classifPos = 1
result = []
run = classif[classifPos]
gen = gen_samples[classifPos]

for i, atom in enumerate(gen):
    #3 coords
    temp=[]
    for j,coord in enumerate(atom):
        #la coord va de -1 a 1, entonces aquí se escala de 0 a 1
        minim = scalers[run*3+j].data_min_[i]
        maxim = scalers[run*3+j].data_max_[i]
        coord = (coord + 1) / 2
        temp.append(coord*(maxim-minim) + minim)
    result.append(temp)
result = np.asarray(result)

fig = pyplot.figure()
ax = Axes3D(fig)

x = result[:,0]
y = result[:,1]
```

```
z = result[:,2]

ax.plot(x, y, z)
pyplot.title('DESESCALADO RUN: '+str(run))
pyplot.show()
```

Código 4.4

En este código se escoge una posición al azar dentro de la variable anteriormente definida *classif* lo que devuelve la muestra a *gen* y el *run* al que pertenece a su variable homónima. A continuación, por cada átomo en *gen* se devuelve a su tamaño original utilizando *scalers*; que guarda los valores máximos y mínimos para la coordenada x,y,z para cada *run* por lo que se compone de 120 valores (40x3), y mediante la fórmula inversa para normalizar valores en un rango de -1 a 1.

$$X = \frac{Y+1}{2} \cdot (MAX - MIN) + MIN \qquad (1)$$

Donde:

- X: Valor original
- Y: Valor escalado
- MAX: Máximo entre los valores originales
- MIN: Mínimo entre los valores originales

Por último, se dibuja el resultado.

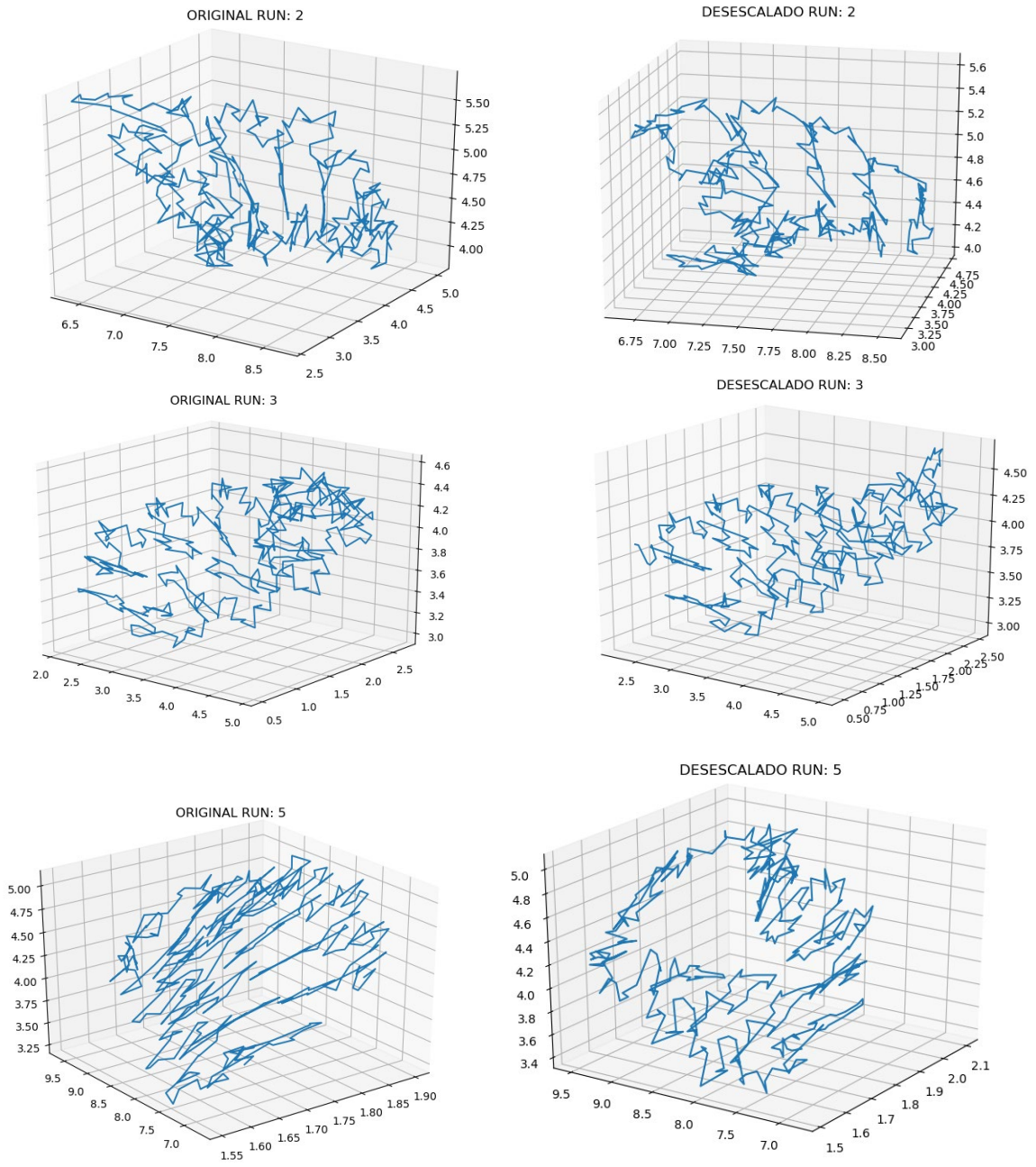


Figura 11. Comparación de muestras generadas con las originales en sus respectivos *runs*

Según la *figura 11*, las conformaciones originales en comparación con las artificiales son visualmente bastante similares. Ahora, observamos el diccionario *normDict* para saber si la distribución de *runs* es equitativo:

Tipo de conformación	Número de coincidencias
0	71
1	53
2	46
3	147
4	134
5	117
6	109
7	41
8	3
9	2
10	13
11	47
12	7
13	9
14	7
15	16
16	13
17	7
18	1
19	21
20	17
21	1
22	7
23	6
24	1
25	18
26	1
27	11
29	24
30	6
31	2
32	3
33	1
34	7
35	5
36	4
37	7
38	7
39	8

Tabla 4. Número de coincidencias por tipo de conformación (*run*)

Se observa como la mayoría de muestras son clasificadas en los primeros *runs* mientras que hay pocas en los últimos. Esto se puede explicar al ver la cantidad de ejemplos que hay en cada tipo de conformación en el conjunto de entrenamiento, como ya se ha mostrado anteriormente:

[23799, 23888, 23877, 48763, 48846, 49001, 49001, 24001, 4001]

Salida 4.2

Parece ser que la GAN genera más conformaciones de un tipo si su número de ejemplos en el conjunto de entrenamiento es mayor que en el de otros. Si se comparan los 8 primeros valores del número de coincidencias de la *tabla 4* con el número de la cantidad de ejemplos en cada *run* queda claro que el tipo de conformación es proporcional a su número de muestras en el entrenamiento.

Si lo que se busca es una GAN que genere conformaciones de cualquiera de los 40 tipos de forma equilibrada habrá que hacer una ligera modificación en el código de entrenamiento.

4.2 Modificaciones en la red

Para conseguir que la red neuronal genere ejemplos de los distintos tipos de *runs* de forma equitativa se entrenará utilizando solamente los primeros 4001 ejemplos de cada tipo de conformación. Justo antes de la parte del código de entrenamiento de la red neuronal que normaliza los datos (*código 3.7*), se implementas esto:

```
        X_eq = []
        jumpIndexesEq = [0]
        for i in range(0,len(jumpIndexes)-1):
            start = jumpIndexes[i]
            end = jumpIndexes[i + 1]
            if (end - start) > 4001:
                X_eq.append(X_train[start:start+4001,:,:])
                jumpIndexesEq.append(jumpIndexesEq[i]+4001)
            else:
                diff = end - start
                X_eq.append(X_train[start:start+diff,:,:])
                jumpIndexesEq.append(jumpIndexesEq[i]+diff)
        X_eq = np.asarray(X_eq)
        X_eq =
X_eq.reshape(X_eq.shape[0]*X_eq.shape[1],X_eq.shape[2],X_eq.shape[3])
#         np.save('C:\\Simulation\\CUC_Na_aligned_jumps_eq.npy',
jumpIndexesEq)
```

```
X_train = X_eq
jumpIndexes = jumpIndexesEq
```

Código 4.5

Es un código que, simplemente, recorta a 4001 todos los *runs* que superan ese tamaño junto con *jumpIndexes* para aplicar posteriormente la normalización de forma correcta.

Después se entrena la red y devuelve un gráfico de función de costes con este aspecto:

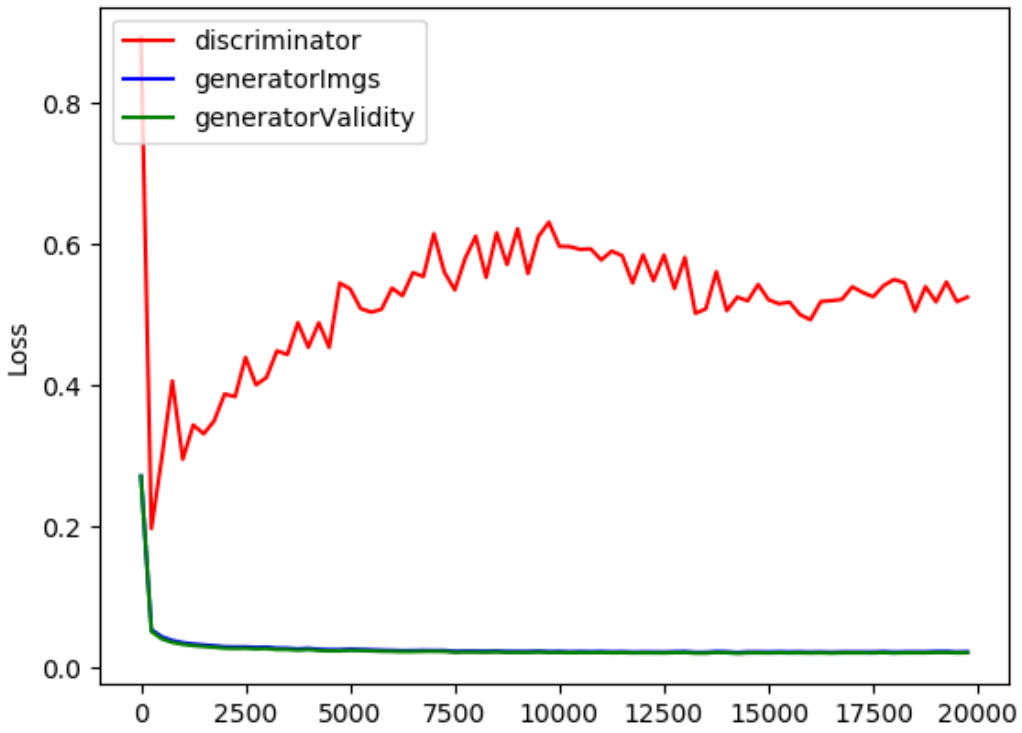


Figura 12. Pérdida en GAN Autoencoder equilibrado

Esta red se ha entrenado en *batchs* de 512 durante 20000 epochs, menos veces que los 40000 que se entrenó la red antes de esta modificación. Esto es debido a que se han eliminado más de la mitad de los ejemplos por la restricción por lo que no era necesario tanto entrenamiento.

La función de coste del *autoencoder* (*generator*) da valores correctos, donde se puede ver que llega al mínimo muy pronto y se mantiene constante por lo que induce a pensar que no hacía falta un entrenamiento tan largo.

4.3 Comparación entre la red neuronal modificada y la original

Se generarán 1000 muestras con la nueva red para proceder a la comparación con la versión antigua. El análisis a esta versión de la GAN se hará de idéntica forma al apartado 4.1.

La siguiente tabla muestra, al igual que la *tabla 4*, el número de coincidencias en las moléculas generadas para cada tipo de conformación o *run*.

Tipo de conformación	Número de coincidencias
0	45
1	11
2	19
3	34
4	45
5	45
6	22
7	18
8	16
9	12
10	36
11	61
12	25
13	32
14	31
15	39
16	39
17	24
18	9
19	34
20	25
21	7
22	14
23	14
24	12
25	31
26	11
27	45

28	17
29	43
30	24
31	9
32	14
33	11
34	12
35	17
36	27
37	42
38	15

Tabla 5. Número de coincidencias por tipo de conformación (*run*). Versión modificada

En comparación con la *tabla 4*, se observa una distribución mucho más equilibrada entre clases; justo lo que se buscaba.

El siguiente paso es comparar el rendimiento usando la media de la variable *mse* para ambas versiones de la red. Como recordatorio, *mse* guarda el valor del *medium squared error* medio con cada uno de los *runs* para cada una de las 1000 muestras artificiales. El resultado se muestra a continuación:

Mse medio versión antigua	Mse medio versión equilibrada
0.096071124	0.09659372

Tabla 6. Número de coincidencias por tipo de conformación (*run*). Versión modificada

El *mse* medio es prácticamente idéntico en ambos casos, por lo que no se puede decir que al eliminar ejemplos de entrenamiento se haya perdido mucha precisión.

Por último, se mostrarán algunas conformaciones generadas por la nueva versión de la red comparadas con muestras reales del mismo *run*; tal y como se hizo en la *figura 9*.

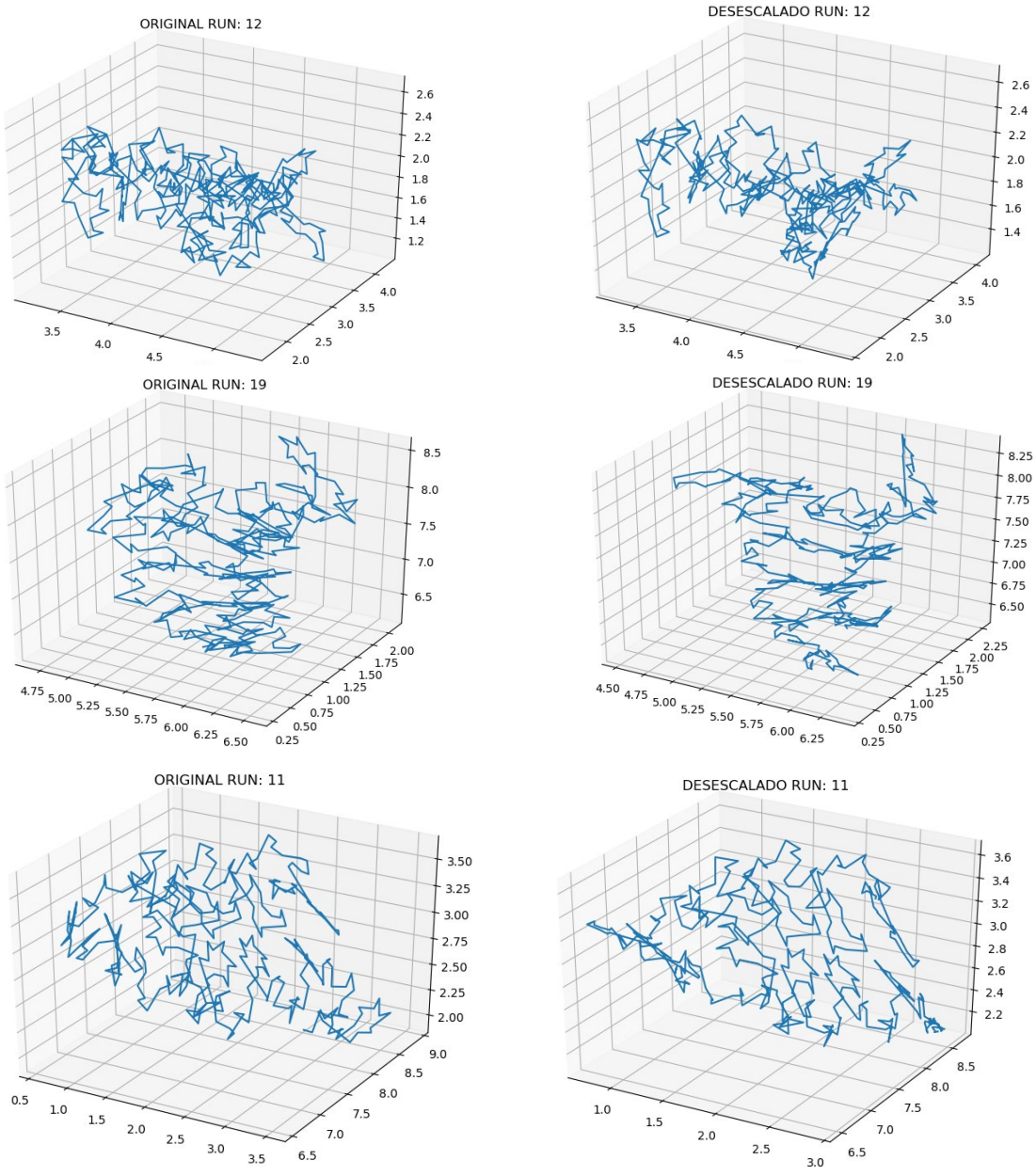


Figura 13. Comparación de muestras generadas con las originales en sus respectivos *runs. Versión equilibrada.*

En la *figura 13* se puede ver como las conformaciones artificiales son muy parecidas a las reales. La calidad de estas figuras es similar a la versión anterior de la red (*figura 11*) y se concluye que la modificación no ha afectado prácticamente nada a las muestras generadas.

5 Clasificación de estados de alta energía

En este apartado se tratará el tema de la clasificación de muestras artificiales o reales como de alta energía libre o no. Aunque este no es el objetivo principal del trabajo, se ha desarrollado un simple sistema que puede ayudar a localizar estas conformaciones específicas.

La idea se basa en cómo se identifican conformaciones de alta energía utilizando histogramas de tres dimensiones en la que se superpondrán todas las conformaciones del conjunto de datos disponibles. Las cuadrículas del histograma con menor número de átomos superpuestos pertenecerán a las conformaciones menos comunes y, por tanto, a las de mayor energía libre.

Sabiendo esto, tal vez se pueda crear algún método que permita averiguar cuáles son las muestras menos comunes y que no requiera de histogramas que pueden ser engorrosos de configurar como el tamaño de la cuadrícula y sus dimensiones, además de que puede darse el caso de que no haya dos conformaciones repetidas; lo que conlleva que todas sean tratadas como del mismo estado energético.

Mediante *mse* (*medium squared error*), como se ha discutido en anteriores apartados, se puede medir cuál de las conformaciones de entrada (generadas por la red o reales) es menos común. La implementación se discute a continuación.

5.1 Implementación del método

Se creará un método que devolverá en orden decreciente de rareza las posiciones o índices en la lista de las conformaciones de entrada.

```
def get_higher_energy_idxs_v11(gen_samples, top):
    #Versión numba

        memoryArray =
np.zeros((len(gen_samples),len(gen_samples)),dtype=np.float32)

    loop_v11_parallel(gen_samples, memoryArray)

    mseMedias = np.mean(memoryArray,axis=1)

    return (-mseMedias).argsort()[:top]
```

Código 5.1

Este método requiere la lista de conformaciones a procesar *gen_samples,* donde cada elemento es la molécula de 343 átomos con sus respectivas 3 coordenadas por cada uno; siempre y cuando pertenezcan al mismo tipo de conformación o *run*. La variable *top* es el número máximo de índices que

devolverá la función. Dentro de la misma se encuentra *memoryArray*, una matriz 2D que sirve para ahorrar tiempo de procesado (como se verá más adelante) y para el *mse* medio de cada molécula en relación a las demás. La última parte importante es la variable que se devuelve, la cual es una lista de las medias *mse* calculadas de mayor a menor.

Ahora se definirá la parte más importante del método: la función *loop_v11_parallel*; que se encarga de calcular los valores de la *memoryArray*.

```
@nb.njit(nb.types.void(nb.float32[:,:,:],nb.float32[:,:]),parallel=True)
def loop_v11_parallel(gen_samples, memoryArray):
    for i in nb.prange(len(gen_samples)):
        for j in range(len(gen_samples)):
            if (memoryArray[i,j] == 0):
                if i != j:
                    #Se suman
                    suma = np.sum((gen_samples[i]-
gen_samples[j])**2,axis=1)
                    #Escoge los 6 más altos y hace media
                    ind = (-suma).argsort()[:6]
                    memoryArray[j,i] = suma[ind].mean()
            else:
                memoryArray[j,i] = memoryArray[i,j]
```

Código 5.2

Esta función utiliza la librería *numba* [7] que permite una mayor eficiencia que solo con la librería *numpy*. Utiliza la cabecera *@nb.njit* y, además, junto con *nb.prange*; permite ejecutar el método en paralelo utilizando todos los núcleos del procesador, lo que divide el tiempo de ejecución dependiendo de los núcleos que hayan disponibles.

En cuanto al contenido del método, recorre una a una todas las moléculas comparándolas con todas las demás, si la variable *memoryArray* es 0 significa que es una comparación que no se había hecho antes; si no que utiliza el valor guardado anteriormente de la comparación antigua. Esa es una de las funciones de *memoryArray*, si no se usara así en el método tardaría hasta el doble de tiempo. Para calcular un valor en *memoryArray*, primero se verifica que no se esté comparando una molécula consigo misma, entonces se calcula la diferencia entre las coordenadas de las dos y se eleva al cuadrado para obtener el *mse*.

El siguiente paso es sumar las coordenadas para cada átomo para, de esta manera, obtener un vector de 343 átomos donde se escogen los 6 más altos, es decir, los átomos de la primera molécula con más diferencia respecto a los mismos átomos de la segunda. Por último, se hace una media de estos 6 valores y se aplican a la posición de *memoryArray* que corresponda.

En resumen, *memoryArray* contiene el *mse* de los 6 átomos con más diferencia de cada molécula de entrada comparada con todas las demás. Para una entrada *gen_samples* de 4001 moléculas tendría un aspecto como este:

	0	1	2	3	4	5	6	7	8
0	0	1.23233	1.17362	1.34932	1.64822	2.17061	2.33101	4.00626	3.5829
1	1.23233	0	0.972718	1.24932	1.60015	2.04777	2.43907	3.41544	2.80769
2	1.17362	0.972718	0	0.893589	1.25439	1.03764	1.83658	2.40103	1.94137
3	1.34932	1.24932	0.893589	0	0.819043	1.65719	2.19485	3.06254	2.18111
4	1.64822	1.60015	1.25439	0.819043	0	1.17638	1.39343	2.06978	1.2428
5	2.17061	2.04777	1.03764	1.65719	1.17638	0	0.891759	1.66747	1.57397
6	2.33101	2.43907	1.83658	2.19485	1.39343	0.891759	0	1.10294	1.05102
7	4.00626	3.41544	2.40103	3.06254	2.06978	1.66747	1.10294	0	1.35048
8	3.5829	2.80769	1.94137	2.18111	1.2428	1.57397	1.05102	1.35048	0

Figura 14. Aspecto aproximado de *memoryArray*.

Evidentemente, la matriz es de un tamaño de 4001x4001 pero solamente se muestran las 9 primeras entradas.

Una intuición de cómo se lee la *figura 14* para que quede más claro sería así: desde la columna 0 (la muestra 0) en comparación con la fila 0 (muestra 0) no se lleva ninguna diferencia, como el lógico. Pero al ir comparando la columna 0 con las filas 1, 2, 3… se está mostrando el *mse* que se lleva la muestra 0 con las muestras 1, 2, 3… hasta la 4000. Entonces no hay más que calcular la media por cada columna (o fila ya que la matriz es simétrica) para obtener las muestras con un valor más alto o, dicho de otro modo, las menos comunes de todas.

5.2 Rendimiento

Un problema de este método es que el número de muestras a procesar incidirá de forma cuadrática al tiempo necesario para completarse o la cantidad de memoria RAM debido a los *for* anidados y a las dimensiones de la matriz respectivamente. También hay que tener en cuenta que cuantos más núcleos tenga el procesador más rápido se ejecutará.

Nº de muestras	Tiempo (4 núcleos) (s)	Tiempo (8 núcleos) (s)	RAM (MB)
4000	35	17,5	61

8000	140	70	244
16000	560	280	976
32000	2240	1120	3904
64000	8960	4480	15616

Tabla 6. Tabla de rendimiento para crear *memoryArray*

Como se deduce de la tabla, para extraer las conformaciones menos comunes de un conjunto de 4000 muestras, el algoritmo hace su trabajo en un tiempo aceptable, pero si se busca en un conjunto de 64000, se hace prácticamente inviable en un ordenador doméstico; por tiempo y por RAM.

5.3 Análisis de resultados

En esta sección se comentarán los resultados al aplicar el algoritmo en muestras reales y en muestras generadas por la red neuronal.

Antes de nada, se definirá el procedimiento de análisis que se llevará a cabo para evaluar la correcta clasificación de las conformaciones. El código completo se encuentra en el *anexo 5.1.*

Cuando se evalúe un conjunto de muestras se presentará una salida generada por el siguiente código:

```
inds = []
sumComps = []
medias = []
for item in idxs:
    sample = item
    gen = X_train[sample]
    mseSoloItem = []
    for item in X_train:
            suma = np.sum((gen-item)**2,axis=1)
        #Escoge los 6 más altos y hace media
        ind = np.argpartition(suma, -6)[-6:]
        mseSoloItem.append(np.take(suma, ind).mean())

    tempus = np.where(mseSoloItem==np.amin(mseSoloItem))
    mseSoloItem = np.delete(mseSoloItem, tempus)
    tempus = int(np.where(mseSoloItem==np.amin(mseSoloItem))[0][0])

    #Ahora se sacan los átomos con más mse
    mseAtoms = (X_train[tempus] - gen)**2
```

```
    mseAtomsSum = np.sum(mseAtoms,axis=1)
    medias.append(np.mean(mseAtomsSum))
    #Ahora se escogen los idx de los atoms con más mse
    topSize = 6#El tamaño del "podium"
    ind = np.argpartition(mseAtomsSum, -topSize)[-topSize:]
    inds.append(ind)

    sumComp = []
    for item in ind:
        atomo = item
        #Ahora se sacan los átomos con más mse
        mseAtomsComp = (X_train[:,atomo,:] - gen[atomo,:])**2
        mseAtomsSumComp = np.sum(mseAtomsComp,axis=1)

        sumComp.append(mseAtomsSumComp.mean())
    sumComps.append(sumComp)

#Ahora sacar los máximos y hacer ránking posibilidades de High energy
maxs = []
for item in sumComps:
    maxs.append(np.sum(item))#Con esto es por la suma de los átomos
maxs.sort(reverse=True)
sumComps = np.asarray(sumComps)
wheres = []
for item in maxs:
    wheres.append(np.where(np.sum(sumComps,axis=1)==item)[0])

#print results
for item in wheres:
    print("index:",idxs[item[0]])
    indexes = np.argsort(-sumComps[item[0]])
    show = sumComps[item[0]][indexes]
    print("Mse per Atom:",show)
    print("Mse suma:",np.sum(show))
    print("Mse media:",medias[item[0]])
    show = inds[item[0]][indexes]
    print("Atom idxs:",show)
    print("----------------------")
```

Código 5.3

Este es un código que simplemente compara las conformaciones poco comunes en los índices obtenidos en *idxs* con todas las demás en *X_train* y obtiene la que tenga un *mse* más bajo comparando los 6 átomos que se explicaron anteriormente. La salida obtenida es la siguiente:

```
index: 3878
Mse per Atom: [3.8289392 3.4706495 3.3372784 1.6690334 1.6056998 1.567819
]
Mse suma: 15.479419
Mse media: 0.27617326
Atom idxs: [272 274 271 150 183 151]
--------------------
index: 3598
Mse per Atom: [3.1856825 2.9012835 2.7254062 2.285086  1.8437021
1.4652814]
Mse suma: 14.406442
Mse media: 0.6576643
Atom idxs: [199 105 197 116 112 110]
```

Salida 5.1. Salida de ejemplo

Para un conjunto de 4001 muestras, *salida 5.1* presenta los índices de las dos conformaciones con más posibilidades de ser de alta energía libre. Para obtener los valores se ha comparado con la muestra con la cual se lleva menos *mse* en relación a sus 6 átomos más "únicos". *Index* representa el índice de la conformación; *Mse per atom* es el *mse* calculado por átomo de mayor a menor; *Mse suma* es la suma de *mse per atom*, y se utiliza para clasificar qué conformación es la menos común (o de alta energía libre); *mse media* es el *mse* medio que tiene la conformación en relación a la muestra que más se le parece en sus 6 átomos "únicos". Por último, *atom idxs* son las posiciones de esos 6 átomos dentro de la conformación.

Sabiendo esto, se dibujarán las conformaciones y se marcarán estos átomos comparándolos con conformaciones adyacentes el tiempo o parecidas para comprobar la consistencia del método.

5.3.1 Análisis de muestras reales

Analizando un *run* de 4001 ejemplos del archivo *CUC_Mg_aligned.npy* previamente depurado de errores da el siguiente resultado:

```
index: 3878
Mse per Atom: [3.8289392 3.4706495 3.3372784 1.6690334 1.6056998 1.567819
]
Mse suma: 15.479419
Mse media: 0.27617326
Atom idxs: [272 274 271 150 183 151]
--------------------
index: 3598
Mse per Atom: [3.1856825 2.9012835 2.7254062 2.285086  1.8437021
1.4652814]
```

```
Mse suma: 14.406442
Mse media: 0.6576643
Atom idxs: [199 105 197 116 112 110]
---------------------
index: 2419
Mse per Atom: [3.215339  3.0868993 2.821493  2.3341455 1.7384138 0.704404
]
Mse suma: 13.900695
Mse media: 0.3151833
Atom idxs: [305 289 287 286 248 189]
---------------------
index: 3425
Mse per Atom: [3.1707652 3.1496387 2.1387222 1.9015249 1.5073072 1.221876
]
Mse suma: 13.089834
Mse media: 0.21968056
Atom idxs: [21 20 50 36 35  3]
---------------------
index: 2689
Mse per Atom: [2.967487  2.940767  2.2427237 1.952105  1.5581189
1.2823774]
Mse suma: 12.943579
Mse media: 1.1301818
Atom idxs: [335 320 135  83  82  80]
```

Salida 5.1. Salida de *CUC_Mg*.

Como la muestra con un *mse* más alto en sus átomos es la número 3878, se comparará con otras muestras para determinar si de verdad es una conformación poco común. Ya que los átomos más destacados están en un rango del 270 a 280, por claridad solo se mostrará esto en los siguientes gráficos.

Se dibujarán las 3 conformaciones anteriores a la 3878 y las 3 posteriores, además de dos muestras alejadas para poder apreciar la evolución de ese segmento en concreto.

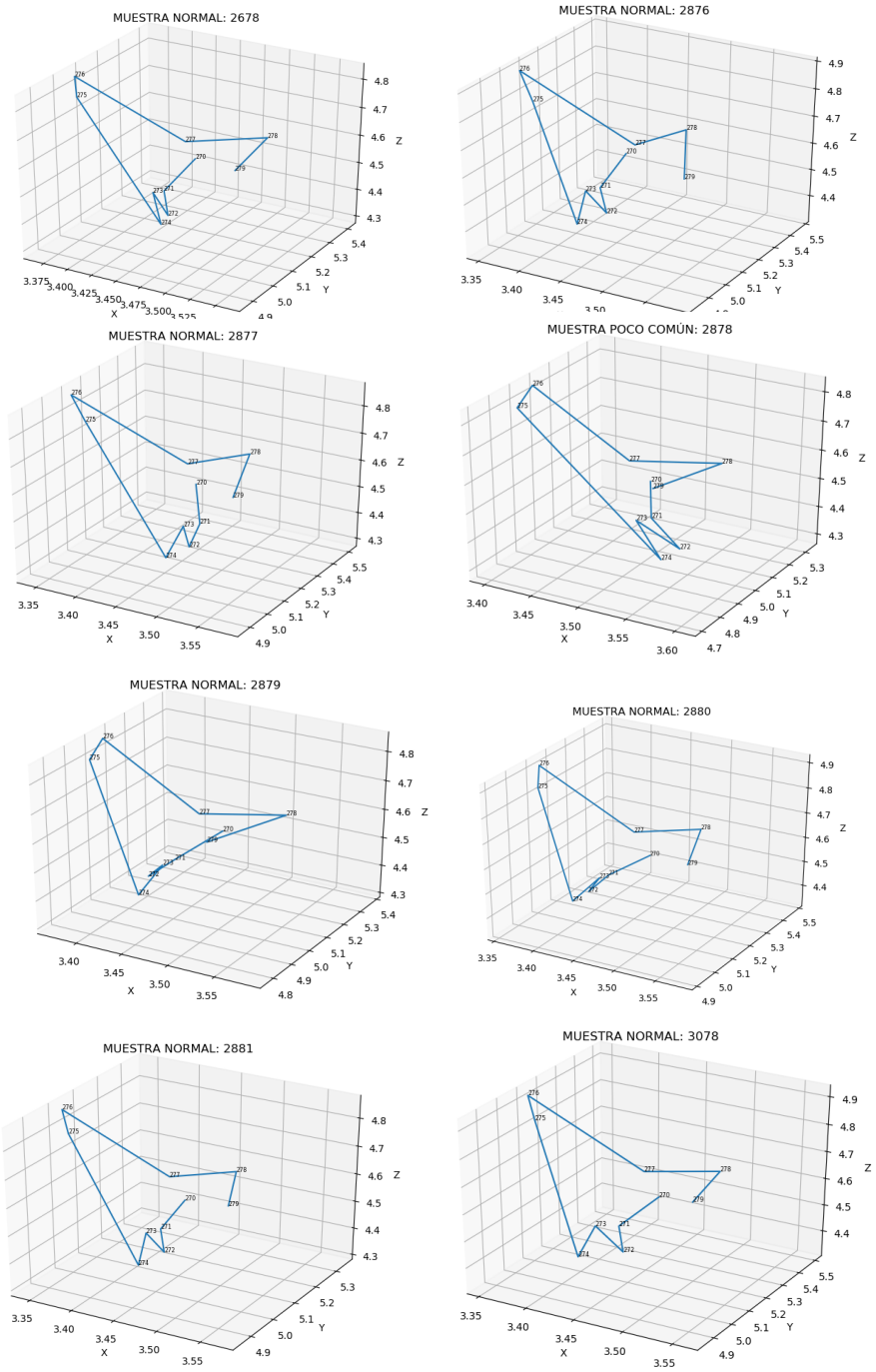


Figura 15. Comparación de segmentos de conformaciones

Se aprecia como el segmento de la muestra 2878 es apreciablemente diferente a los demás. Se puede observar como la dos muestra immediatemente posteriores entran en una especie de plegamiento que parece plano para después volver a un estado común. Las demás muestras son semejantes, incluso la primera y la última que están más alejadas en el tiempo (2678 y 3078 respectivamente) son muy parecidas a las más comunes.

En este caso parece que el algoritmo ha clasificado bien como posible estado de alta energía la muestra 2878 entre los átomos 270 y 280.

5.3.2 Análisis de muestras artificiales

En este caso se seguirá un procedimiento muy similar al del apartado anterior para el archivo *CUC_Na_aligned.npy*, con unas ligeras diferencias. La primera es que para que el algoritmo funcione correctamente, las muestras han de ser de un solo *run*; esto no se puede controlar en las muestras generadas por lo que hay que clasificarlas previamente. Lo siguiente a tener en cuenta es que el algoritmo funciona más lento si el *run* a clasificar es muy grande (mucho más de 4000 muestras), por lo que habría que recortar cada uno al tamaño deseado antes.

Una vez preparado, no hay más que generar unas 160000 muestras. Se requiere este número de muestras ya que, si se buscan unas 4000 de media por tipo de conformación, sabiendo que hay unos 40 tipos de conformación posibles se requieren: $4000 \cdot 40 = 160000$.

Por último, se ejecuta el código y muestra la siguiente salida:

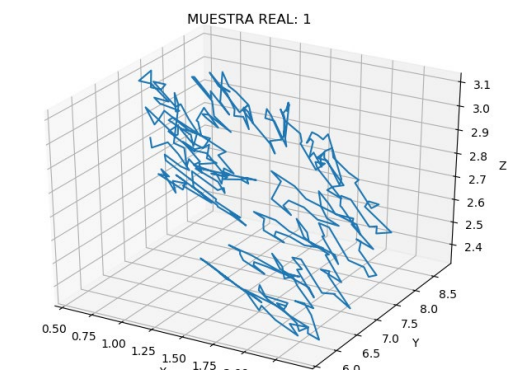
```
============ New run:0 =============

Position: 0 Size: 4001
Mse per Atom: [0.80722314 0.75851935 0.69794106 0.6767218  0.5896627
0.5895766 ]
Atom idxs: [161 330 154 329 328 327]
---------------------
Position: 1 Size: 4001
Mse per Atom: [0.47198138 0.46926853 0.4649775  0.4648257  0.43306512
0.42798698]
Atom idxs: [ 32 137  31  34  35  37]
---------------------
============ New run:1 =============

Position: 0 Size: 1793
Mse per Atom: [0.6149017  0.4866146  0.4688099  0.42421275 0.39429733
0.2987521 ]
Atom idxs: [ 24  26  28 175  92  93]
---------------------
Position: 1 Size: 1793
Mse per Atom: [0.87583315 0.42668927 0.3990794  0.3821597  0.31109214
0.23143181]
Atom idxs: [ 32 340 291 292 293 304]
---------------------
```

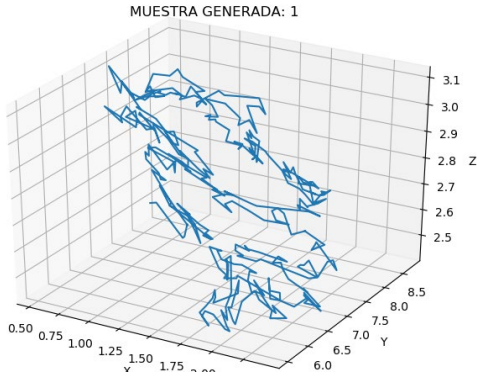
Salida 5.2. Salida de *CUC_Na*.

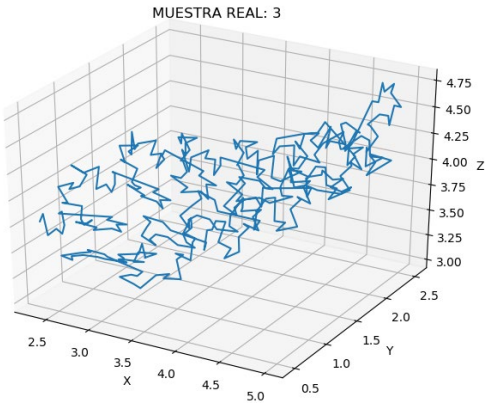
En esta salida es un poco diferente a la *Salida 5.1* ya que clasifica varios *runs* distintos a la vez, por lo que primero muestra los *Mse per Atom* de las dos conformaciones menos comunes del *run* 0; después las dos menos comunes del *run* 1; así hasta llegar a los 40.

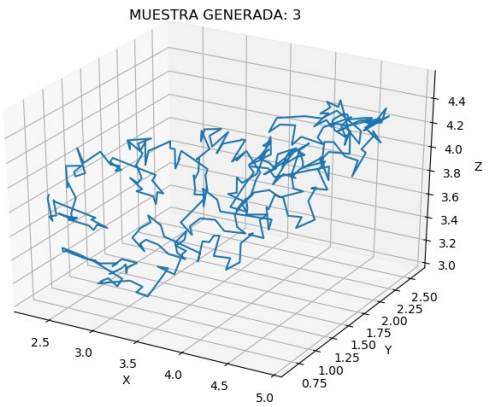
Lo que se observa es que los valores en *Mse per Atom* para la *Salida 5.2* en las muestras generadas no son tan altos como los valores en la *Salida 5.1* de las muestras reales. La conclusión que se puede extraer de esto hecho es que la red neuronal genera una proporción más alta de conformaciones comunes de las que podría haber en un conjunto de muestras reales.

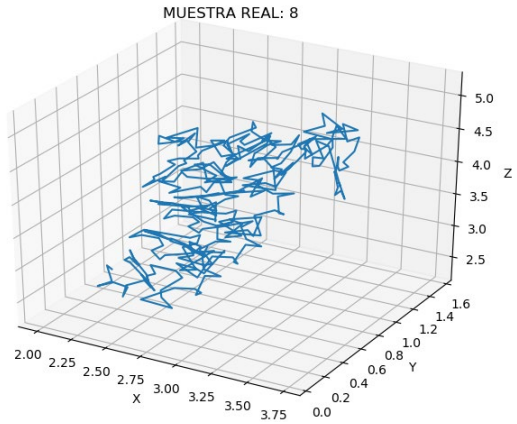
A continuación, se mostrarán una lista de las muestras generadas clasificadas como de alta energía (al menos, presumiblemente) en comparación con *runs* reales de su misma clase.

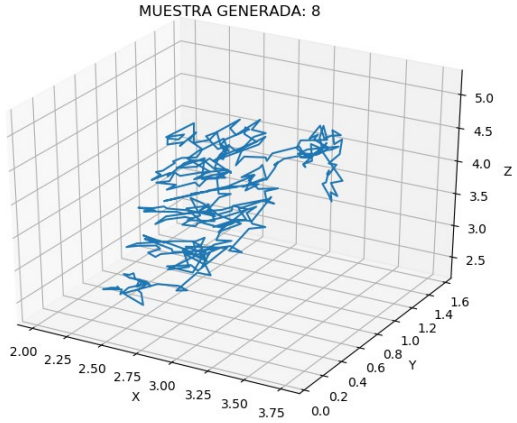

MUESTRA REAL: 1
X
Y
Z

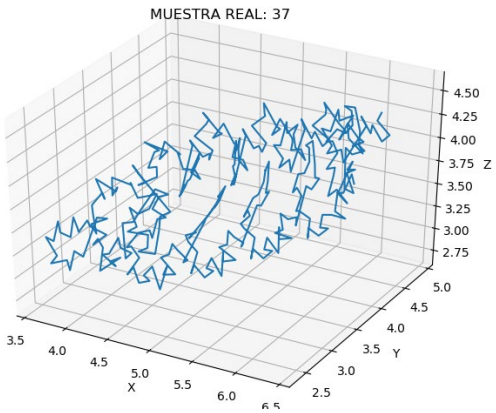

MUESTRA GENERADA: 1
X
Y
Z


MUESTRA REAL: 3
X
Y
Z


MUESTRA GENERADA: 3
X
Y
Z


MUESTRA REAL: 8
X
Y
Z


MUESTRA GENERADA: 8
X
Y
Z


MUESTRA REAL: 37
X
Y
Z

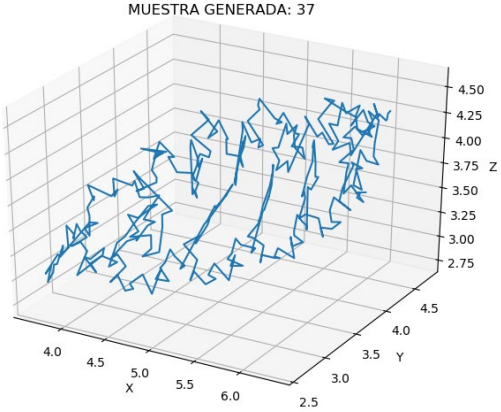

MUESTRA GENERADA: 37
X
Y
Z

Figura 16. Comparación de conformaciones artificiales de, presumiblemente, alta energía con muestras reales.

En la *figura* 16, estas conformaciones artificiales supuestamente de alta energía muestran mucha semejanza con sus contrapartidas reales, aunque es difícil saber si realmente son poco comunes o no, según la *salida 5.2* donde los valores *mse per Atom* eran más bajos que en las muestras reales, lo más probable es que no lo sean.

En resumen, se puede decir que el algoritmo es un buen método para clasificar muestras reales, donde es más probable que haya conformaciones poco comunes; pero es más difícil de establecer para las muestras artificiales porque en general se crean formas comunes.

Conclusiones

La red neuronal es capaz de generar conformaciones muy semejantes a sus tipos de conformación reales, pero es difícil decir si estas muestras generadas son útiles en un laboratorio sin la opinión de un experto. Además, se ha podido crear un método para depurar los errores en los conjuntos de entrenamiento. En general, esta GAN es capaz de generar nuevas conformaciones que pueden ayudar a explorar mejor los posibles plegamientos del ARN ahorrando las semanas de simulación necesarias para obtener las muestras reales.

En cuanto al objetivo extra, el algoritmo de clasificación de muestras como de alta energía libre parece prometedor aplicado a conjuntos de conformaciones reales, siempre y cuando no sean demasiado grandes. En muestras artificiales no ha podido detectar conformaciones de muy alta energía, posiblemente debido a la naturaleza de la red en generar muestras comunes. Aun así, haría falta poner mucho más a prueba la fiabilidad de este algoritmo, lo que no ha sido posible en este trabajo por una cuestión de tiempo.

Bibliografía

1. Wrzalek, Sandro. 2017. "Mg2+ Dependent Folding Behavior Of RNA". Freie Universität Berlin.
2. "Welcome To Python.Org". 2019. *Python.Org*. https://www.python.org/.
3. "Home - Keras Documentation". 2019. *Keras.Io*. https://keras.io/.
4. Linder-Norén, Erik. 2019. "Eriklindernoren/Keras-GAN". *Github*. https://github.com/eriklindernoren/Keras-GAN/blob/master/aae/aae.py.
5. Makhzani, Alireza, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Fey. 2016. "Adversarial Autoencoders". *Corr*. doi:1511.05644.
6. Ruder, Sebastian. 2016. "An Overview Of Gradient Descent Optimization Algorithms". *Corr* abs/1609.04747. doi:1609.04747.
7. "Numba: A High Performance Python Compiler". 2019. *Numba.Pydata.Org*. https://numba.pydata.org/.
8. "Numpy — Numpy". 2019. *Numpy.Org*. https://www.numpy.org/.
9. "A Beginner's Guide To Neural Networks And Deep Learning". 2019. *Skymind*. https://skymind.ai/wiki/neural-network.
10. "A Beginner's Guide To Generative Adversarial Networks (Gans)". 2019. *Skymind*. https://skymind.ai/wiki/generative-adversarial-network-gan.

Anexo

1. Comparación de los dos métodos de cálculo de distancias.

Se compararán los dos métodos según las gráficas de distancias medias cada 50 ejemplos dibujadas a partir de la variable *noInf_dist* filtrada de valores erróneos como ya se explica en el apartado *2.2.1 Identificando problemas*.

Variante: `np.sum((p1-p2)**2)`

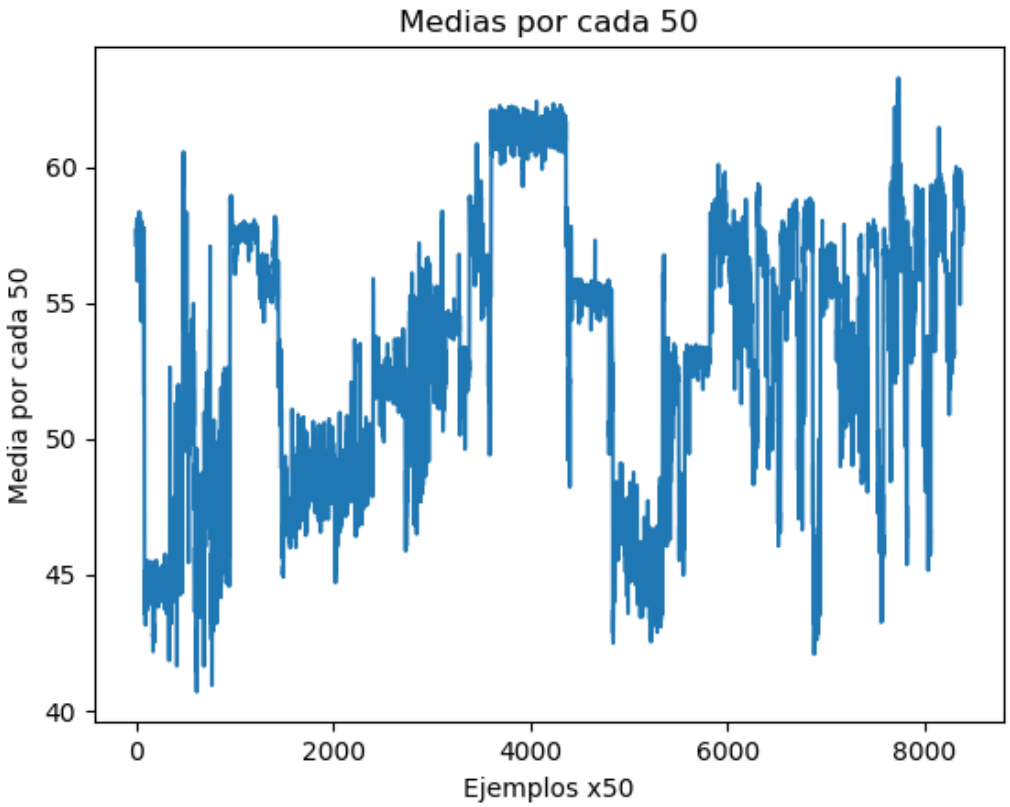


Figura 17. Medias de distancia filtradas variante 1

Variante: `np.sum(p1**2 + p2**2)`

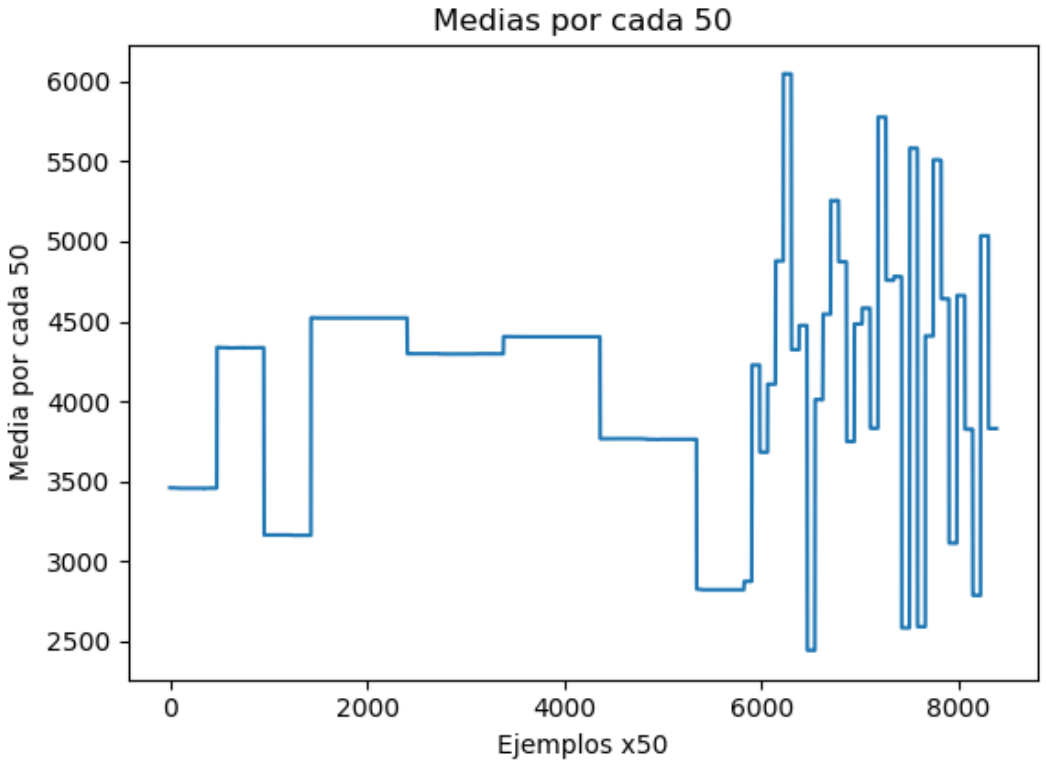


Figura 18. Medias de distancia filtradas variante 2

A causa de elevar al cuadrado ambos átomos en la segunda variante, las diferencias se vuelvan tan altas que permite dibujar una gráfica sin apenas "ruido", al contrario de la primera variante.

2. Código completo del capítulo 2: Depurando los datos

```
#Calcular tamaño medio de la molécula si estuviera estirada para los 10
primeros ejemplos (comprobando que estos sean correctos)

import numpy as np
from numpy import inf
import MedirTiempo as tmp
import matplotlib.pyplot as plt

data = np.load('C:\Simulation\CUC_Mg_aligned.npy', encoding='bytes')
distancesPath = 'C:\Simulation\CUC_Mg_aligned_distances_old.npy'
loadDistances = True

distance = []

#    C:\Simulation\CUC_Mg_aligned_distances.npy
```

```
if(not loadDistances):
    tmp.time_counter(init=True)
    for j, example in enumerate(data):
        dist = [0,0]#EL primero la distancia, el segundo la desviación
estándard
        forstd = []
        for i in range(1,len(example)):#Empieza en uno
            p1 = np.array([example[i-1][0], example[i-1][1], example[i-
1][2]])
            p2 = np.array([example[i][0], example[i][1], example[i][2]])

#             squared_dist = np.sum((p1-p2)**2, axis=0)
            squared_dist = np.sum(p1**2 + p2**2, axis=0)
            forstd.append(squared_dist)
            dist[0] += np.sqrt(squared_dist)
        dist[1] = np.std(forstd)
        distance.append(dist)
        tmp.time_counter(examplesLength=len(data),currentExample=j)
    distance = np.asarray(distance)
    np.save(distancesPath, distance)
else:
    distance = np.load(distancesPath)

def printMeansAndCo(noInf_dist):
    print("Valor medio máximo:",np.amax(noInf_dist[:,0]))
    print("Posición del valor medio máximo",np.where(noInf_dist ==
np.amax(noInf_dist[:,0])))
    print("Valor medio mínimo:",np.amin(noInf_dist[:,0]))
    print("Desviación estándard máxima:",np.amax(noInf_dist[:,1]))
    print("Posición del std máximo",np.where(noInf_dist ==
np.amax(noInf_dist[:,1])))

def drawGraphs(chunk_size, noInf_dist):
    def chunks(a, chunk_size):
        counter = 0
        output = []
        if(chunk_size > 0 and chunk_size <= len(a)):
            while counter+chunk_size <= len(a):
                output.append(a[counter:counter+chunk_size])
                counter += chunk_size
            if(counter < len(a)):
                output.append(a[counter:len(a)])

        return output

    chunkos = chunks(noInf_dist, chunk_size)
```

```
    means = []
    for item in chunkos:
        temp = [0,0]
        temp[0] = np.mean(item[:,0])
        temp[1] = np.std(item[:,1])
        means.append(temp)
    means = np.asarray(means)
    plt.figure()
    plt.plot(list(range(0,len(means))), means[:,0])
    plt.xlabel('Ejemplos x'+str(chunk_size))
    plt.ylabel('Media por cada '+str(chunk_size))
    plt.title("Medias por cada "+str(chunk_size))
    plt.figure()
    plt.plot(list(range(0,len(means))), means[:,1])
    plt.xlabel('Ejemplos x'+str(chunk_size))
    plt.ylabel('Std media por cada '+str(chunk_size))
    plt.title("Std por cada "+str(chunk_size))
    plt.show()

    print("Valor de data máximo:",np.amax(data))
    print("Valor de data mínimo:",np.amin(data))

printMeansAndCo(distance)
#Aqui se eliminan los infinitos generados
data = data[distance[:,0] < inf]
noInf_dist = distance[distance[:,0] < inf]

printMeansAndCo(noInf_dist)

"""
Si la desviación estándar o la media según las gráficas de más abajo
son demasiado grandes
es que, posiblemente, hay errores de medida. Por ejemplo, en el caso
CUC_Mg
se ven picos en las gráficas entorno a los ejemplos 2400 y 800. Entonces
se filtrarán los datos con una media estándar superior a 1000 y se
volverá a analizar para ver si sigue habiendo errores y se iterará
"""

"""
MANUAL
"""
chunk_size = 50#Modificar esta variable manualmente
"""
FIN MANUAL
"""
```

```
"""
ESTO DE ABAJO ES MANUAL
"""

stdFilterValue = 1000
#Con este valor (1000) se filtra la media estándar de los ejemplos con
#errores como se puede ver
#si se elimina las líneas de abajo y se ejecuta el código otra vez
data = data[noInf_dist[:,1] < stdFilterValue]
noInf_dist = noInf_dist[noInf_dist[:,1] < stdFilterValue]

printMeansAndCo(noInf_dist)

drawGraphs(chunk_size, noInf_dist)
"""
FIN MANUAL
"""

"""
Si se quieren crear histogramas diferenciando la molécula en distintas
etapas o medidas hay que explorar means manualmente y ver dónde
varían mucho los elementos de la columna "0" (distancias) y apuntar los
índices. Se puede usar la gráfica de Media por cada 50 como para ver
cuál es el salto mínimo
y entonces obtener índices donde los saltos sean igual o mayores a ese
salto.
"""
"""
Según la gráfica el salto mínimo es 25 así que se sacarán índices donde
el salto sea superior a 15 (por asegurar) y se compararán los índices
con la gráfica a ver si encajan
"""
"""
MANUAL
"""
minimumJumpLength = 15
"""
FIN MANUAL
"""
jumpIndexes = []
for i in range(1, len(noInf_dist)):
    if(abs(noInf_dist[i,0] - noInf_dist[i-1,0]) > minimumJumpLength):
        jumpIndexes.append(i)

"""
```

```
Si se mira la diferencia entre números consecutivos de jumpNumber se ve
un patrón
"""
#Añadir el 0 y el último para que se muestren todos los intervalos
jumpIndexes = [0] + jumpIndexes + [len(noInf_dist)]
np.save('C:\Simulation\CUC_Mg_aligned_jumps', jumpIndexes)
difference =  np.ediff1d(jumpIndexes)
```

3. Código completo del capítulo 3: Creación de la red neuronal

```
from __future__ import print_function, division

from keras.datasets import mnist
from keras.layers import Input, Dense, Reshape, Flatten, Dropout
from keras.layers import BatchNormalization, Activation, ZeroPadding1D,
Conv2DTranspose
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling1D, Conv1D
from keras.models import Sequential, Model,  model_from_json
from keras.optimizers import Adam
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from keras.layers import Lambda

import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

import keras.backend as K

import numpy as np


class DCGAN():
    def __init__(self):
         #Necesario porque va yendo cada vez más lento
        K.clear_session()

        # Input shape
        self.num_atoms = 343#Son los átomos por muestra
        self.dimensions = 3#Las coordenadas por átomo
        self.sample_shape = (self.num_atoms, self.dimensions)
        self.latent_dim = 343#Las dimensiones a las cuales reducir el
input
```

```
        self.loss = []

        self.optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
            optimizer=self.optimizer,
            metrics=['accuracy'])

        # Build the encoder / decoder
        self.encoder = self.build_encoder()
        self.decoder = self.build_decoder()

        img = Input(shape=self.sample_shape)
        # Se codifica la conformación y después se recontruye
        encoded_repr = self.encoder(img)
        reconstructed_img = self.decoder(encoded_repr)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator determines validity of the encoding
        validity = self.discriminator(encoded_repr)

        #Significa que habrán dos outputs, reconstr_img y validity
        #Y sus contribuciones a la función de coste 0,999 y 0,001
respectivamente
        #Para los tipos de función de coste está mse (mean-squared-error)
        #y binary_crossentropy
        self.adversarial_autoencoder = Model(img, [reconstructed_img,
validity])
        self.adversarial_autoencoder.compile(loss=['mse',
'binary_crossentropy'],
            loss_weights=[0.999, 0.001],
            optimizer=self.optimizer)

    def build_encoder(self):
        # Encoder

        img = Input(shape=self.sample_shape)

        h = Flatten()(img)
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
```

```
        h = Dropout(0.25)(h)
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
        h = Dropout(0.25)(h)
        h = Dense(1024)(h)

        h = LeakyReLU(alpha=0.2)(h)
        h = Dropout(0.25)(h)

        #Esto de abajo depende del paper:
https://arxiv.org/pdf/1511.05644.pdf
        mu = Dense(self.latent_dim)(h)
        log_var = Dense(self.latent_dim)(h)

        latent_repr = Lambda(lambda p: p[0] +
K.random_normal(K.shape(p[0])) * K.exp(p[1] / 2))([mu, log_var])

        return Model(img, latent_repr)

    def build_decoder(self):

        model = Sequential()

        model.add(Dense(1024, input_dim=self.latent_dim))

        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Dense(1024))

        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))
        model.add(Dense(1024))

        model.add(LeakyReLU(alpha=0.2))
        model.add(Dropout(0.25))

        model.add(Dense(np.prod(self.sample_shape), activation='tanh'))
        model.add(Reshape(self.sample_shape))

        model.summary()

        z = Input(shape=(self.latent_dim,))
        img = model(z)

        return Model(z, img)
```

```
    def build_discriminator(self):

        model = Sequential()

        model.add(Dense(512, input_dim=self.latent_dim))

        model.add(LeakyReLU(alpha=0.2))

        model.add(Dense(256))

        model.add(LeakyReLU(alpha=0.2))

        model.add(Dense(1, activation="sigmoid"))
        model.summary()

        encoded_repr = Input(shape=(self.latent_dim, ))
        validity = model(encoded_repr)

        return Model(encoded_repr, validity)

    def train(self, epochs, batch_size=128, loss_interval=50,
save_interval=1000):

        subName = "all"
        subFolderName = "AEv2-Eq"
        folder = "C:/Simulation/savedModels/%s/" % subFolderName

        X_train = np.load('C:\\Simulation\\CUC_Na_aligned.npy',
encoding='bytes')
        #jumpIndexes que te dicen donde están los saltos entre runs
        jumpIndexes = np.load('C:\\Simulation\\CUC_Na_aligned_jumps.npy',
encoding='bytes')
#
        #Ahora hay que escoger sólo los 4001 primeros ejemplos de cada
run
        X_eq = []
        jumpIndexesEq = [0]
        for i in range(0,len(jumpIndexes)-1):
            start = jumpIndexes[i]
            end = jumpIndexes[i + 1]
            if (end - start) > 4001:
                X_eq.append(X_train[start:start+4001,:,:])
                jumpIndexesEq.append(jumpIndexesEq[i]+4001)
            else:
                diff = end - start
```

```
                X_eq.append(X_train[start:start+diff,:,:])
                jumpIndexesEq.append(jumpIndexesEq[i]+diff)
        X_eq = np.asarray(X_eq)
        X_eq =
X_eq.reshape(X_eq.shape[0]*X_eq.shape[1],X_eq.shape[2],X_eq.shape[3])
#         np.save('C:\\Simulation\\CUC_Na_aligned_jumps_eq.npy',
jumpIndexesEq)

        X_train = X_eq
        jumpIndexes = jumpIndexesEq


        minAndMaxValues = []

        scaler = MinMaxScaler(feature_range=(-1, 1))
        for j in range(0,len(jumpIndexes)-1):
            minMaxItem = []
            start = jumpIndexes[j]
            end = jumpIndexes[j + 1]
            for i in range(X_train.shape[2]):
                subItem = []
                subItem.append(np.amin(X_train[start:end, :, i]))
                subItem.append(np.amax(X_train[start:end, :, i]))
                minMaxItem.append(subItem)
                X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])

            minAndMaxValues.append(minMaxItem)

        # Adversarial ground truths
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))


        for epoch in range(epochs):

            # ---------------------
            #  Train Discriminator
            # ---------------------

            # Se seleccionan conformaciones reales al azar
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            conf = X_train[idx]

            # Aquí se genera ruido e imágenes reales
            latent_fake = self.encoder.predict(conf)
```

```
            latent_real = np.random.normal(size=(batch_size,
self.latent_dim))

            # Train the discriminator (real classified as ones and
generated as zeros)
            d_loss_real = self.discriminator.train_on_batch(latent_real,
valid)
            d_loss_fake = self.discriminator.train_on_batch(latent_fake,
fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            # ---------------------
            #  Train Generator
            # ---------------------

            # el autoencoder busca que las conformaciones las tome el
discriminator como falsas
            g_loss = self.adversarial_autoencoder.train_on_batch(conf,
[conf, valid])

            # Plot the progress
            print ("%d [D loss: %f, acc: %.2f%%] [G loss: %f, mse: %f]" %
(epoch, d_loss[0], 100*d_loss[1], g_loss[0], g_loss[1]))


            if epoch % loss_interval == 0:
                # Plot the progress
                self.loss.append([epoch, d_loss[0], g_loss[0],
g_loss[1]])
            # If at save interval => save generated image samples
            if epoch % save_interval == 0:
                #Guardar modelo a JSON (para yaml cargar libreria
model_from_yaml el resto es igual
                #Pero cambiando json por yaml)
                #25MB de media
                if(not math.isnan(g_loss[0]) and not
math.isnan(g_loss[1])):
                    model_discriminator = self.discriminator.to_json()
                    model_AE = self.adversarial_autoencoder.to_json()

                    with
open(folder+"modelAE_discriminator_chkpoint.json", "w") as json_file:
                        json_file.write(model_discriminator)
                    with open(folder+"modelAE_AE_chkpoint.json", "w") as
json_file:
                        json_file.write(model_AE)
```

```
                    #Guardar weights a HDF5

self.discriminator.save_weights(folder+"modelAE_discriminator_chkpnt.h5")

self.adversarial_autoencoder.save_weights(folder+"modelAE_AE_cjkpnt.h5")
                    print("Guardado!")

        #Se generan mil samples con el noise de 100 de largo
        noise = np.random.normal(size=(1000, self.latent_dim))
        #Si generator solo acepta noise como un vector de tamaño 100, ¿
como le estas
        #metiendo un noise de tamaño (half,100)? porque en predict, noise
puede ser sólo tamaño 100
        #Si solo le vas a meter un input, pero si le vas a meter un batch
se acepta como una matriz
        #de tamaño (batchsize,100)
        gen_samples = self.decoder.predict(noise)

np.save('C:\\Simulation\\savedModels\\%s\\CUC_Na_aligned_GAN_gen_%s.npy'
% (subFolderName, subName), gen_samples)

np.save('C:\\Simulation\\savedModels\\%s\\CUC_Na_aligned_MinMaxValues_%s.
npy' % (subFolderName, subName), np.asarray(minAndMaxValues))


        #Guardar modelo a JSON (para yaml cargar libreria model_from_yaml
el resto es igual
        #Pero cambiando json por yaml)
        model_discriminator = self.discriminator.to_json()
        model_AE = self.adversarial_autoencoder.to_json()

        with open(folder+"modelAE_discriminator_%s.json" % subName, "w")
as json_file:
            json_file.write(model_discriminator)
        with open(folder+"modelAE_AE_%s.json" % subName, "w") as
json_file:
            json_file.write(model_AE)

        #Guardar weights a HDF5

self.discriminator.save_weights(folder+"modelAE_discriminator_%s.h5" %
subName)

self.adversarial_autoencoder.save_weights(folder+"modelAE_AE_%s.h5" %
subName)
```

```
        print("Guardado!")


        return self.loss, subName

if __name__ == '__main__':
    dcgan = DCGAN()
    returned = dcgan.train(epochs=20000, batch_size=512,
loss_interval=250, save_interval=2000)
    loss = returned[0]
    loss = np.asarray(loss)
    fig = plt.figure()
    plt.plot(loss[:,0], loss[:,1], 'r', label="discriminator") # plotting
t, a separately
    plt.plot(loss[:,0], loss[:,2], 'b', label="generatorImgs") # plotting
t, b separately
    plt.plot(loss[:,0], loss[:,3], 'g', label="generatorValidity") #
plotting t, b separately
    plt.ylabel('Loss')
    plt.legend(loc='upper left')
    plt.show()
    fig.savefig("images/lossGANv2EqAuto_%s.png" % returned[1])
```

4. Código completo del capítulo 4: Análisis de resultados

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler

X_train = np.load('C:\\Simulation\\CUC_Na_aligned.npy', encoding='bytes')
#jumpIndexes que te dicen donde están los saltos entre runs
jumpIndexes = np.load('C:\\Simulation\\CUC_Na_aligned_jumps.npy',
encoding='bytes')

encoding='bytes')

folder = "AEv2"

folderEq = "AEv2-Eq"


X_eq = []
jumpIndexesEq = [0]
for i in range(0,len(jumpIndexes)-1):
    start = jumpIndexes[i]
```

```
    end = jumpIndexes[i + 1]
    if (end - start) > 4001:
        X_eq.append(X_train[start:start+4001,:,:])
        jumpIndexesEq.append(jumpIndexesEq[i]+4001)
    else:
        diff = end - start
        X_eq.append(X_train[start:start+diff,:,:])
        jumpIndexesEq.append(jumpIndexesEq[i]+diff)
X_eq = np.asarray(X_eq)
X_eq =
X_eq.reshape(X_eq.shape[0]*X_eq.shape[1],X_eq.shape[2],X_eq.shape[3])

X_train_eq = X_eq

#Para poder hacer el desescalado luego
scalers = []

for j in range(0,len(jumpIndexes)-1):

    start = jumpIndexes[j]
    end = jumpIndexes[j + 1]
    for i in range(X_train.shape[2]):
        scaler = MinMaxScaler(feature_range=(-1, 1))
        X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])
        scalers.append(scaler)

scalersEq = []

for j in range(0,len(jumpIndexesEq)-1):

    start = jumpIndexesEq[j]
    end = jumpIndexesEq[j + 1]
    for i in range(X_train_eq.shape[2]):
        scalerEq = MinMaxScaler(feature_range=(-1, 1))
        X_train_eq[start:end, :, i] =
scalerEq.fit_transform(X_train_eq[start:end, :, i])
        scalersEq.append(scalerEq)

#Hacer media de X_train para calcular diferencias de mse MUCHO más rápido
X_train_mean = []
for j in range(0,len(jumpIndexes)-1):
    tempTrain = X_train[jumpIndexes[j]:jumpIndexes[j+1]]
    X_train_mean.append(np.mean(tempTrain,axis=0))
X_train_mean = np.asarray(X_train_mean)
```

```
X_train_mean_eq = []
for j in range(0,len(jumpIndexesEq)-1):
    tempTrain = X_train_eq[jumpIndexesEq[j]:jumpIndexesEq[j+1]]
    X_train_mean_eq.append(np.mean(tempTrain,axis=0))
X_train_mean_eq = np.asarray(X_train_mean_eq)

#MANUAL AQUÍ SE CAMBIA EL ARCHIVO DE GEN PARA LAS DIFERENTES PRUEBAS
gen_samples =
np.load('C:\\Simulation\\savedModels\\%s\\CUC_Na_aligned_GAN_gen_all.npy'
% folder, encoding='bytes')

mse = []
for i, gen in enumerate(gen_samples):

    mseItem = []
    mseMean = []
    for item in X_train_mean:
        mseItem.append(((gen - item)**2).mean())
    for j in range(0,len(jumpIndexes)-1):
        mseMean.append(np.mean(mseItem[j:j+1]))
    mse.append(mseMean)
classif = []
for item in mse:
    classif.append(int(np.where(item==np.amin(item))[0]))
normDict = {i:classif.count(i) for i in classif}

mseMeans = []
for item in mse:
    mseMeans.append(np.mean(item))
print(np.mean(mseMeans))

gen_samples_eq =
np.load('C:\\Simulation\\savedModels\\%s\\CUC_Na_aligned_GAN_gen_all.npy'
% folderEq, encoding='bytes')

mseEq = []

for i, gen in enumerate(gen_samples_eq):
    mseEqItem = []
    mseEqMean = []
    for item in X_train_mean_eq:
        mseEqItem.append(((gen - item)**2).mean())
    for j in range(0,len(jumpIndexesEq)-1):
        mseEqMean.append(np.mean(mseEqItem[j:j+1]))

    mseEq.append(mseEqMean)
```

```
classifEq = []
for item in mseEq:
    classifEq.append(int(np.where(item==np.amin(item))[0]))
EqDict = {i:classifEq.count(i) for i in classifEq}

mseEqMeans = []
for item in mseEq:
    mseEqMeans.append(np.mean(item))
print(np.mean(mseEqMeans))

#Por SEPARADO

from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D
import random

"""SE CAMBIAN ESTOS SEIS"""
classifLocal = classifEq
classifPos = 2
gen = gen_samples_eq[classifPos]
scalersLocal = scalersEq
jumpIndexesLocal = jumpIndexesEq
X_train_local = X_train_eq
"""HASTA AQUÍ"""


result = []
run = classifLocal[classifPos]


for i, atom in enumerate(gen):
    #3 coords
    temp=[]
    for j,coord in enumerate(atom):
        #la coord va de -1 a 1, entonces aquí se escala de -1 a 1
        minim = scalersLocal[run*3+j].data_min_[i]
        maxim = scalersLocal[run*3+j].data_max_[i]
        coord = (coord + 1) / 2
        temp.append(coord*(maxim-minim) + minim)
    result.append(temp)
result = np.asarray(result)

fig = pyplot.figure()
ax = Axes3D(fig)
```

```
x = result[:,0]
y = result[:,1]
z = result[:,2]

ax.plot(x, y, z)
pyplot.title('DESESCALADO RUN: '+str(run))
pyplot.show()

fig = pyplot.figure()
ax = Axes3D(fig)


result = []
sample_num = random.randint(jumpIndexesLocal[run],
jumpIndexesLocal[run+1])
gen = X_train_local[sample_num]

for i, atom in enumerate(gen):
    #3 coords
    temp=[]
    for j,coord in enumerate(atom):
        #la coord va de -1 a 1, entonces aquí se escala de 0 a 1
        minim = scalersLocal[run*3+j].data_min_[i]
        maxim = scalersLocal[run*3+j].data_max_[i]
        coord = (coord + 1) / 2
        temp.append(coord*(maxim-minim) + minim)
    result.append(temp)
result = np.asarray(result)

x = result[:,0]
y = result[:,1]
z = result[:,2]

ax.plot(x, y, z)
pyplot.title('ORIGINAL RUN: '+str(run))
pyplot.show()
```

5. Código del capítulo 5: Clasificación de estados de alta energía

5.1 Código para muestras reales

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from matplotlib import pyplot
```

```
from mpl_toolkits.mplot3d import Axes3D
import numba as nb

X_train = np.load('C:\\Simulation\\CUC_Mg_aligned_X.npy',
encoding='bytes')
#jumpIndexes que te dicen donde están los saltos entre runs
jumpIndexes = np.load('C:\\Simulation\\CUC_Mg_jump_idxs.npy',
encoding='bytes')

jumpIndexes = jumpIndexes[10:12]

folder1 = "AEv2-Eq"
folder2 = "AE-Eq"

X_eq = []
jumpIndexesEq = [jumpIndexes[0]]
for i in range(0,len(jumpIndexes)-1):
    start = jumpIndexes[i]
    end = jumpIndexes[i + 1]
    if (end - start) > 4001:
        X_eq.append(X_train[start:start+4001,:,:])
        jumpIndexesEq.append(jumpIndexesEq[i]+4001)
    else:
        diff = end - start
        X_eq.append(X_train[start:start+diff,:,:])
        jumpIndexesEq.append(jumpIndexesEq[i]+diff)
X_eq = np.asarray(X_eq)
X_eq =
X_eq.reshape(X_eq.shape[0]*X_eq.shape[1],X_eq.shape[2],X_eq.shape[3])

X_train = X_eq
X_train_org = X_eq.copy()
jumpIndexes = jumpIndexesEq
jumpIndexes[:] = jumpIndexes[:] - jumpIndexes[0]

minAndMaxValues = []

#Para poder hacer el desescalado luego
scalers = []

for j in range(0,len(jumpIndexes)-1):

    minMaxItem = []
    start = jumpIndexes[j]
    end = jumpIndexes[j + 1]
    for i in range(X_train.shape[2]):
```

```
        scaler = MinMaxScaler(feature_range=(-1, 1))
        subItem = []
        subItem.append(np.amin(X_train[start:end, :, i]))
        subItem.append(np.amax(X_train[start:end, :, i]))
        minMaxItem.append(subItem)
        X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])
        scalers.append(scaler)
    minAndMaxValues.append(minMaxItem)


@nb.njit(nb.types.void(nb.float32[:,:,:],nb.float32[:,:]),parallel=True)
def loop_v11_parallel(gen_samples, memoryArray):
    for i in nb.prange(len(gen_samples)):
        for j in range(len(gen_samples)):
            if (memoryArray[i,j] == 0):
                if i != j:
                    #Se suman
                    suma = np.sum((gen_samples[i]-
gen_samples[j])**2,axis=1)
                    #Escoge los 6 más altos y hace media
                    ind = (-suma).argsort()[:6]
                    memoryArray[j,i] = suma[ind].mean()
            else:
                memoryArray[j,i] = memoryArray[i,j]

def get_to_delete_and_minimvalue_v11(gen_samples, top):

    memoryArray =
np.zeros((len(gen_samples),len(gen_samples)),dtype=np.float32)

    loop_v11_parallel(gen_samples, memoryArray)

    mseMedias = np.mean(memoryArray,axis=1)

    return (-mseMedias).argsort()[:top]

#Aquí se clasifica el output
temp_samples = X_train

idxs =  get_to_delete_and_minimvalue_v11(temp_samples, 20)

inds = []
sumComps = []
medias = []
for item in idxs:
```

```
    sample = item
    gen = X_train[sample]
    mseSoloItem = []
    for item in X_train:
        suma = np.sum((gen-item)**2,axis=1)
        #Escoge los 6 más altos y hace media
        ind = np.argpartition(suma, -6)[-6:]
        mseSoloItem.append(np.take(suma, ind).mean())

    tempus = np.where(mseSoloItem==np.amin(mseSoloItem))
    mseSoloItem = np.delete(mseSoloItem, tempus)
    tempus = int(np.where(mseSoloItem==np.amin(mseSoloItem))[0][0])

    #Ahora se sacan los átomos con más mse
    mseAtoms = (X_train[tempus] - gen)**2
    mseAtomsSum = np.sum(mseAtoms,axis=1)
    medias.append(np.mean(mseAtomsSum))
    #Ahora se escogen los idx de los atoms con más mse
    topSize = 6#El tamaño del "podium"
    ind = np.argpartition(mseAtomsSum, -topSize)[-topSize:]
    inds.append(ind)


    sumComp = []
    for item in ind:
    #Según ind cambias este número para comparar
        atomo = item
        #Ahora se sacan los átomos con más mse
        mseAtomsComp = (X_train[:,atomo,:] - gen[atomo,:])**2
        mseAtomsSumComp = np.sum(mseAtomsComp,axis=1)

        sumComp.append(mseAtomsSumComp.mean())
    sumComps.append(sumComp)

#Ahora sacar los máximos y hacer ránking posibilidades de High energy
maxs = []
for item in sumComps:
    maxs.append(np.sum(item))#Con esto es por la suma de los átomos
maxs.sort(reverse=True)
sumComps = np.asarray(sumComps)
wheres = []
for item in maxs:
    wheres.append(np.where(np.sum(sumComps,axis=1)==item)[0])

#print results
for item in wheres:
```

```
    print("index:",idxs[item[0]])
    indexes = np.argsort(-sumComps[item[0]])
    show = sumComps[item[0]][indexes]
    print("Mse per Atom:",show)
    print("Mse suma:",np.sum(show))
    print("Mse media:",medias[item[0]])
    show = inds[item[0]][indexes]
    print("Atom idxs:",show)
    print("----------------------")


#Por SEPARADO

sample = 3878
atomInitPos = 270


def plot_atom_section(sample_num, title, atominitPos, size=10):
    x = X_train_org[sample_num,atomInitPos:atomInitPos+10,0]
    y = X_train_org[sample_num,atomInitPos:atomInitPos+10,1]
    z = X_train_org[sample_num,atomInitPos:atomInitPos+10,2]

    fig = pyplot.figure()
    ax = Axes3D(fig)

    ax.plot(x, y, z)
    pyplot.title(title+": "+str(sample_num))
    ax.set_xlabel("X")
    ax.set_ylabel("Y")
    ax.set_zlabel("Z")
    for x,y,z,i in zip(x,y,z,range(atomInitPos, atomInitPos+len(x))):
        ax.text(x,y,z,i,fontsize='xx-small')
    pyplot.show()
```

5.1 Código para muestras reales

```
from keras.models import model_from_json
import numpy as np
from keras.optimizers import Adam
import random
import time
import numba as nb


@nb.njit(nb.types.void(nb.float32[:,:,:],nb.float32[:,:]),parallel=True)
```

```
def loop_v11_parallel(gen_samples, memoryArray):
    for i in nb.prange(len(gen_samples)):
        for j in range(len(gen_samples)):
            if (memoryArray[i,j] == 0):
                if i != j:
                    #Se suman
                    suma = np.sum((gen_samples[i]-
gen_samples[j])**2,axis=1)
                    #Escoge los 6 más altos y hace media

                    ind = (-suma).argsort()[:6]
                    memoryArray[j,i] = suma[ind].mean()
            else:
                memoryArray[j,i] = memoryArray[i,j]

def get_higher_energy_idxs_v11(gen_samples, top=-1):
    #Versión numba
    #4000 Ejemplos serán unos 61mb, 8000 244 y así
    #4000 en 4 núcleos de cpu tarda 35s, para 8000 140s y así
    #Para 64000 samples usaría 2h 30 min y 15,6 gb

    memoryArray =
np.zeros((len(gen_samples),len(gen_samples)),dtype=np.float32)

    loop_v11_parallel(gen_samples, memoryArray)

    mseMedias = np.mean(memoryArray,axis=1)

    if(top < 1 or top > len(gen_samples)):
        top = len(gen_samples)
    return (-mseMedias).argsort()[:top]

def filter_gen_list(gen_samples_list, max_num_samples):
    for i, gen_samples in enumerate(gen_samples_list):
        if(len(gen_samples) >= max_num_samples):
            #Para que no se repitan
            a = np.arange(len(gen_samples))
            np.random.shuffle(a)
            idx = a[:max_num_samples]
            gen_samples = gen_samples[idx]
        gen_samples_list[i] = gen_samples
    return gen_samples_list

def classify(gen_samples, classificator, number_of_runs):
    classificator = np.asarray(classificator)
    gen_samples_list = []
```

```
    for i in range(number_of_runs):
        wheres = np.where(classificator==i)[0]
        if(len(wheres) > 0):
            gen_samples_list.append(gen_samples[wheres])
    return gen_samples_list

def restore_size(gen, run):
    result = []
    for i, atom in enumerate(gen):
        #3 coords
        temp=[]
        for j,coord in enumerate(atom):
            #la coord va de -1 a 1, entonces aquí se escala de -1 a 1
            minim = scalers[run*3+j].data_min_[i]
            maxim = scalers[run*3+j].data_max_[i]
            coord = (coord + 1) / 2
            temp.append(coord*(maxim-minim) + minim)
        result.append(temp)
    return result
def get_classificator(temp_samples,run_size,rootName):
    from sklearn.preprocessing import MinMaxScaler

    X_train = np.load('C:\\Simulation\\%s_aligned.npy'%rootName,
encoding='bytes')
    #jumpIndexes que te dicen donde están los saltos entre runs
    jumpIndexes =
np.load('C:\\Simulation\\%s_aligned_jumps.npy'%rootName,
encoding='bytes')

    #Si folder1 Y 2 son -Eq descomentar esto
    X_eq = []
    jumpIndexesEq = [0]
    for i in range(0,len(jumpIndexes)-1):
        start = jumpIndexes[i]
        end = jumpIndexes[i + 1]
        if (end - start) > run_size:
            X_eq.append(X_train[start:start+run_size,:,:])
            jumpIndexesEq.append(jumpIndexesEq[i]+run_size)
        else:
            diff = end - start
            X_eq.append(X_train[start:start+diff,:,:])
            jumpIndexesEq.append(jumpIndexesEq[i]+diff)
    X_eq = np.asarray(X_eq)
    X_eq =
X_eq.reshape(X_eq.shape[0]*X_eq.shape[1],X_eq.shape[2],X_eq.shape[3])
```

```
    #         np.save('C:\\Simulation\\CUC_Na_aligned_jumps_eq.npy',
jumpIndexesEq)

    X_train = X_eq
#     X_train_org = X_eq.copy()
    jumpIndexes = jumpIndexesEq


    #Para poder hacer el desescalado luego
    scalers = []

    for j in range(0,len(jumpIndexes)-1):

        start = jumpIndexes[j]
        end = jumpIndexes[j + 1]
        for i in range(X_train.shape[2]):
            scaler = MinMaxScaler(feature_range=(-1, 1))
            subItem = []
            subItem.append(np.amin(X_train[start:end, :, i]))
            subItem.append(np.amax(X_train[start:end, :, i]))
            X_train[start:end, :, i] =
scaler.fit_transform(X_train[start:end, :, i])
            scalers.append(scaler)


    #Hacer media de X_train para calcula diferencias de mse MUCHO más
rápido
    X_train_mean = []
    for j in range(0,len(jumpIndexes)-1):
        tempTrain = X_train[jumpIndexes[j]:jumpIndexes[j+1]]
        X_train_mean.append(np.mean(tempTrain,axis=0))
    X_train_mean = np.asarray(X_train_mean)


    #MANUAL AQUÍ SE CAMBIA EL ARCHIVO DE GEN PARA LAS DIFERENTES PRUEBAS

    gen_samples_1 = temp_samples

    mseSolo = []
    for i, gen in enumerate(gen_samples_1):

        mseSoloItem = []
        mseSoloMean = []
        for item in X_train_mean:
            mseSoloItem.append(((gen - item)**2).mean())
        for j in range(0,len(jumpIndexes)-1):
```

```
            mseSoloMean.append(np.mean(mseSoloItem[j:j+1]))
        mseSolo.append(mseSoloMean)
    classifUno = []
    for i, item in enumerate(mseSolo):
        classifUno.append(int(np.where(item==np.amin(item))[0][0]))
    soloDict = {i:classifUno.count(i) for i in classifUno}

    mseSoloMeans = []
    for item in mseSolo:
        mseSoloMeans.append(np.mean(item))
    print("Media de error al clasificar: ",np.mean(mseSoloMeans))
    return classifUno, soloDict, scalers

subName = "all"
subFolderName = "AEv2-Eq"
rootName = "CUC_Na"
folder = "C:/Simulation/savedModels/%s/" % subFolderName

run_size = 4001
number_of_runs = 40
latent_dim = 343

optimizer = Adam(0.0002, 0.5)

 #Para cargar modelo json y weights
json_disc = open(folder+"modelAE_discriminator_%s.json" % subName, "r")
loaded_disc_json = json_disc.read()
json_disc.close()
discriminator = model_from_json(loaded_disc_json)
#Para cargar HDF5 (weights)
discriminator.load_weights(folder+"modelAE_discriminator_%s.h5" %
subName)


json_AE = open(folder+"modelAE_AE_%s.json" % subName, "r")
loaded_AE_json = json_AE.read()
json_AE.close()
adversarial_autoencoder = model_from_json(loaded_AE_json)
#Para cargar HDF5 (weights)
adversarial_autoencoder.load_weights(folder+"modelAE_AE_%s.h5" % subName)


adversarial_autoencoder.compile(loss=['mse', 'binary_crossentropy'],
loss_weights=[0.999, 0.001],
optimizer=optimizer)
```

```
discriminator.compile(loss='binary_crossentropy',
    optimizer=optimizer,
    metrics=['accuracy'])

print("Cargado!")

#Aquí se saca el decoder que hay dentro, creo que siempre se llama
model_3
#Porque es el 3 modelo que se generó
decoder = adversarial_autoencoder.get_layer('model_3')

#Se generan 1000 o lo que sea de samples con el noise de 343 de largo
size = run_size*number_of_runs
noise = np.random.normal(size=(size, latent_dim))

gen_samples = decoder.predict(noise)

classifUno, soloDict, scalers =
get_classificator(gen_samples,run_size,rootName)

gen_samples_list = classify(gen_samples, classifUno, number_of_runs)

gen_samples = None

#Por el rendimiento
gen_samples_list = filter_gen_list(gen_samples_list,run_size)

idxs_list = []

for i, gen_samples in enumerate(gen_samples_list):
#Aquí se clasifica el output
    temp_samples = gen_samples
    start = time.time()
    idxs =  get_higher_energy_idxs_v11(temp_samples, 20)
    idxs_list.append(idxs)
    print("Tiempo transcurrido y run:",time.time() - start, i)
gens_to_save = []

f = open(folder+"idxsAndCo.txt","w+")

#Aquí se genera un txt con el resumen de las muestras más raras, sus
átomos, etc.
for j, idxs in enumerate(idxs_list):
    newRun = "============ New run:%d ============\r" % j
    counter = 0
    print(newRun,file=f)
```

```
X_train = gen_samples_list[j]
X_train_resized = []
inds = []
sumComps = []
medias = []
for item in idxs:
    X_train_resized.append(restore_size(X_train[item], j))
    sample = item
    gen = X_train[sample]
    mseSoloItem = []
    for item in X_train:
        suma = np.sum((gen-item)**2,axis=1)
        #Escoge los 6 más altos y hace media
        ind = np.argpartition(suma, -6)[-6:]
        mseSoloItem.append(np.take(suma, ind).mean())

    tempus = np.where(mseSoloItem==np.amin(mseSoloItem))
    mseSoloItem = np.delete(mseSoloItem, tempus)
    tempus = int(np.where(mseSoloItem==np.amin(mseSoloItem))[0][0])

    #Ahora se sacan los átomos con más mse
    mseAtoms = (X_train[tempus] - gen)**2
    mseAtomsSum = np.sum(mseAtoms,axis=1)
    medias.append(np.mean(mseAtomsSum))

    #Ahora se escogen los idx de los atoms con más mse
    topSize = 6#El tamaño del "podium"
    ind = np.argpartition(mseAtomsSum, -topSize)[-topSize:]
    inds.append(ind)


    sumComp = []
    for item in ind:
    #Según ind cambias este número para comparar
        atomo = item
        #Ahora se sacan los átomos con más mse
        mseAtomsComp = (X_train[:,atomo,:] - gen[atomo,:])**2
        mseAtomsSumComp = np.sum(mseAtomsComp,axis=1)

        sumComp.append(mseAtomsSumComp.mean())
    sumComps.append(sumComp)

#Ahora sacar los máximos y hacer ránking posibilidades de High energy
maxs = []
for item in sumComps:
    maxs.append(np.sum(item))#Con esto es por la suma de los átomos
```

```
    maxs.sort(reverse=True)
    sumComps = np.asarray(sumComps)
    wheres = []
    for item in maxs:
        wheres.append(np.where(np.sum(sumComps,axis=1)==item)[0])

    #print results
    body = ""
    for item in wheres:

        print("Position:",counter,"Size:",len(X_train),file=f)
        indexes = np.argsort(-sumComps[item[0]])
        show = sumComps[item[0]][indexes]
        print("Mse per Atom:",show,file=f)
        print("Mse media:",medias[item[0]],file=f)#Una mse media alta
puede
        #ser indicativo de HES, pq si ha tenido que buscar uno muy
diferente
        #para que se parezca a su segmento más diferente es pq no lo ha
encontrado
        #en conformaciones parecidas
        show = inds[item[0]][indexes]
        print("Atom idxs:",show,file=f)
        print("----------------------",file=f)
        counter += 1
    gens_to_save.append(np.asarray(X_train_resized))

f.close()
gens_to_save = np.asarray(gens_to_save)
np.save('C:\\Simulation\\%s_gens_HES.npy'%rootName,gens_to_save)


#Por SEPARADO
X_train = np.load('C:\\Simulation\\%s_aligned.npy'%rootName,
encoding='bytes')
jumpIndexes = np.load('C:\\Simulation\\%s_aligned_jumps.npy'%rootName,
encoding='bytes')

from matplotlib import pyplot
from mpl_toolkits.mplot3d import Axes3D

gens_to_save_idx = 4
subidx = random.randint(0, len(gens_to_save[gens_to_save_idx])-1)
x = gens_to_save[gens_to_save_idx][subidx,:,0]
y = gens_to_save[gens_to_save_idx][subidx,:,1]
z = gens_to_save[gens_to_save_idx][subidx,:,2]
```

```
fig = pyplot.figure()
ax = Axes3D(fig)

ax.plot(x, y, z)
pyplot.title('GEN HES')
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
pyplot.show()

gennormal = gen_samples_list[gens_to_save_idx]
sample_num = random.randint(0, len(gennormal)-1)
result = restore_size(gennormal[sample_num], gens_to_save_idx)
result = np.asarray(result)
fig = pyplot.figure()
ax = Axes3D(fig)
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
x = result[:,0]
y = result[:,1]
z = result[:,2]

ax.plot(x, y, z)
pyplot.title('GEN NORMAL: %d'%sample_num)
pyplot.show()


subidx = random.randint(jumpIndexes[gens_to_save_idx],
jumpIndexes[gens_to_save_idx+1]-1)

x = X_train[subidx,:,0]
y = X_train[subidx,:,1]
z = X_train[subidx,:,2]

fig = pyplot.figure()
ax = Axes3D(fig)

ax.plot(x, y, z)
pyplot.title('REAL')
ax.set_xlabel("X")
ax.set_ylabel("Y")
ax.set_zlabel("Z")
pyplot.show()
```

6. Más comparaciones de conformaciones reales con artificiales

En este apartado del anexo se comparan conformaciones reales con artificiales del mismo *run* representadas con otro método de dibujado. Debido a la falta de tiempo no se ha podido implementar en el resto de figuras de este trabajo que, aunque son comprensibles, no son tan "vistosas".

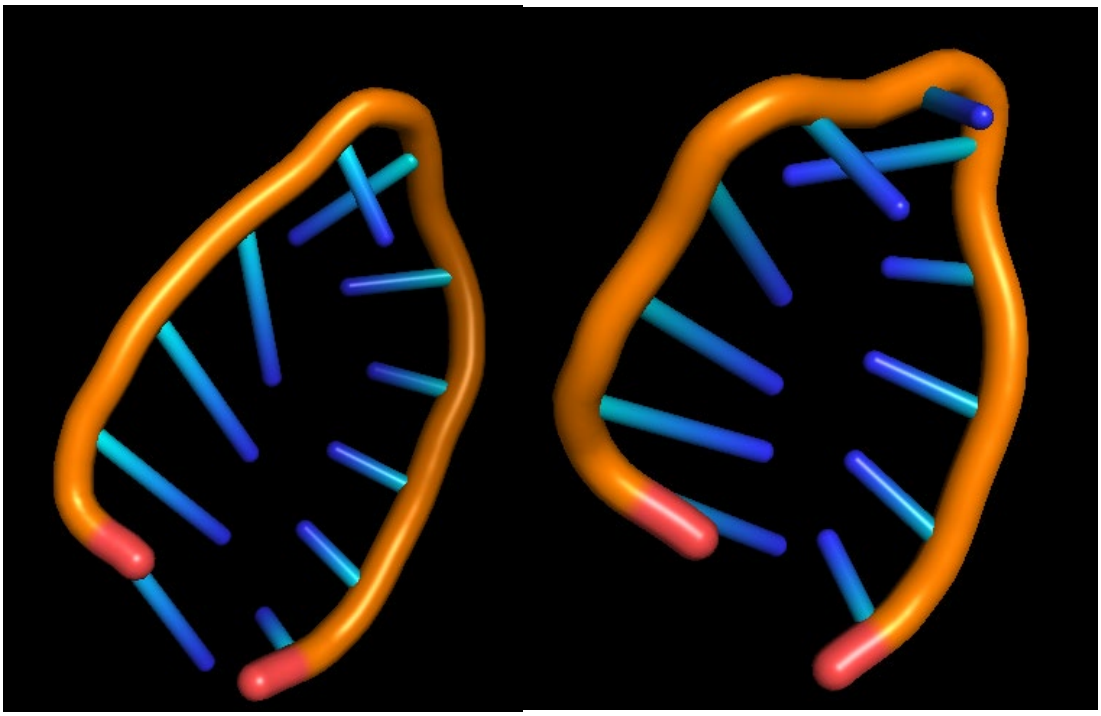


Figura 19. Muestra real (izquierda) y muestra artificial (derecha) del *run* 0

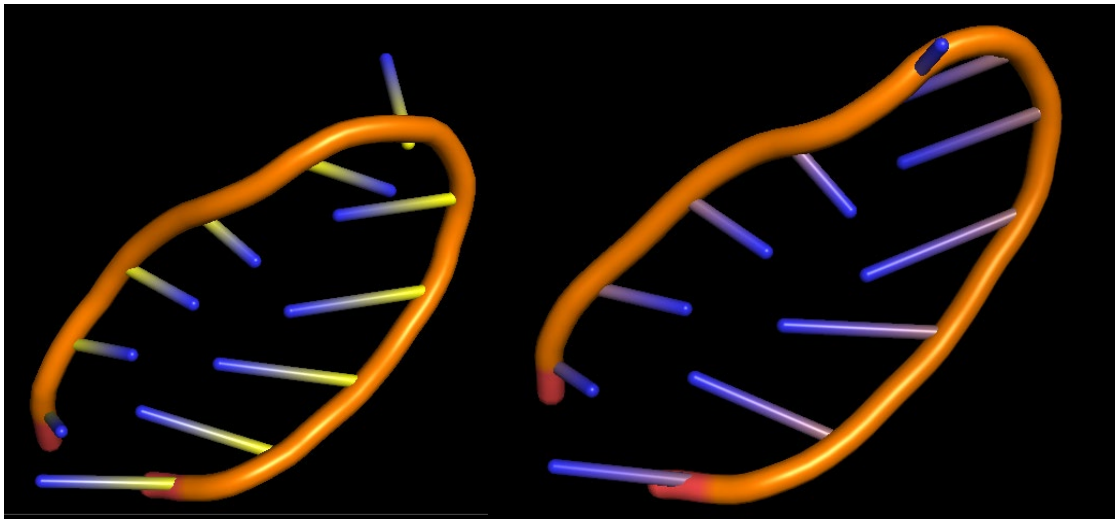


Figura 20. Muestra real (izquierda) y muestra artificial (derecha) del *run* 3

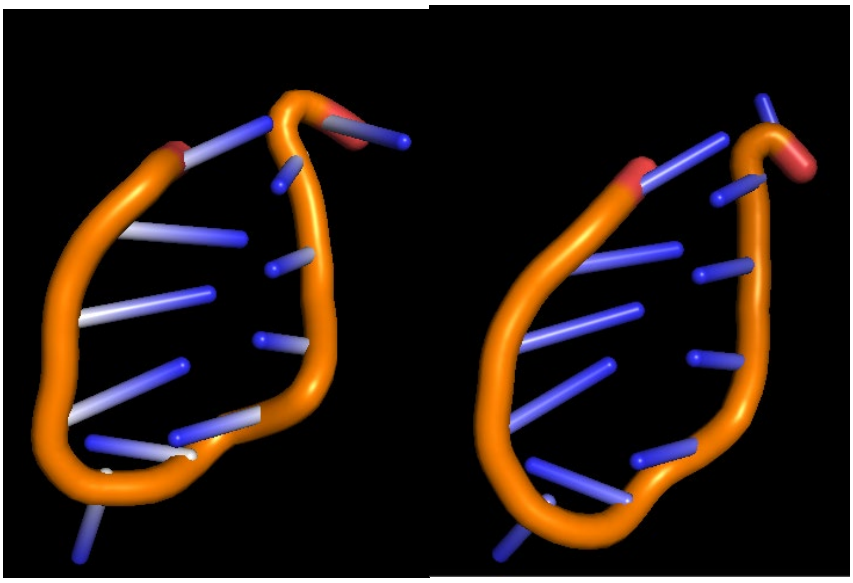


Figura 21. Muestra real (izquierda) y muestra artificial (derecha) del *run* 4

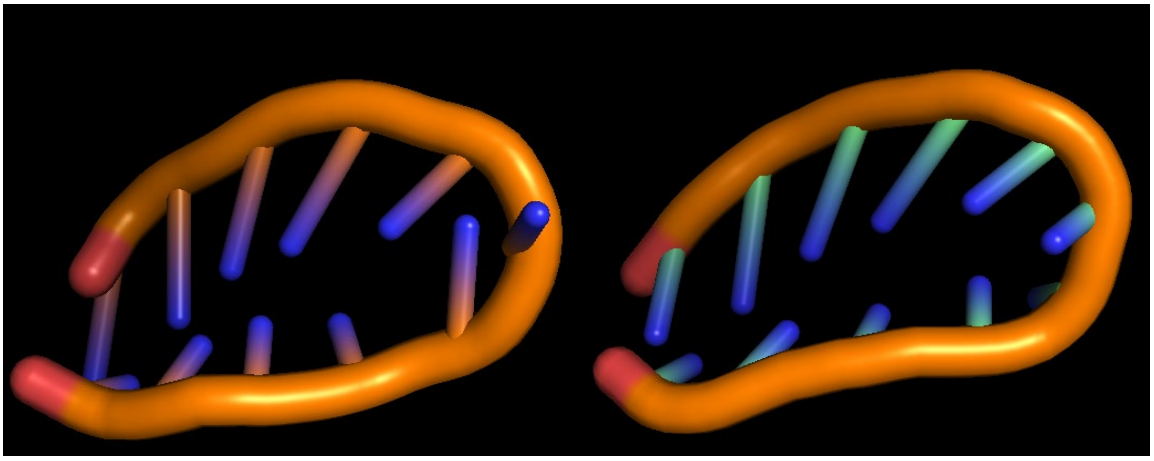


Figura 22. Muestra real (izquierda) y muestra artificial (derecha) del *run* 6

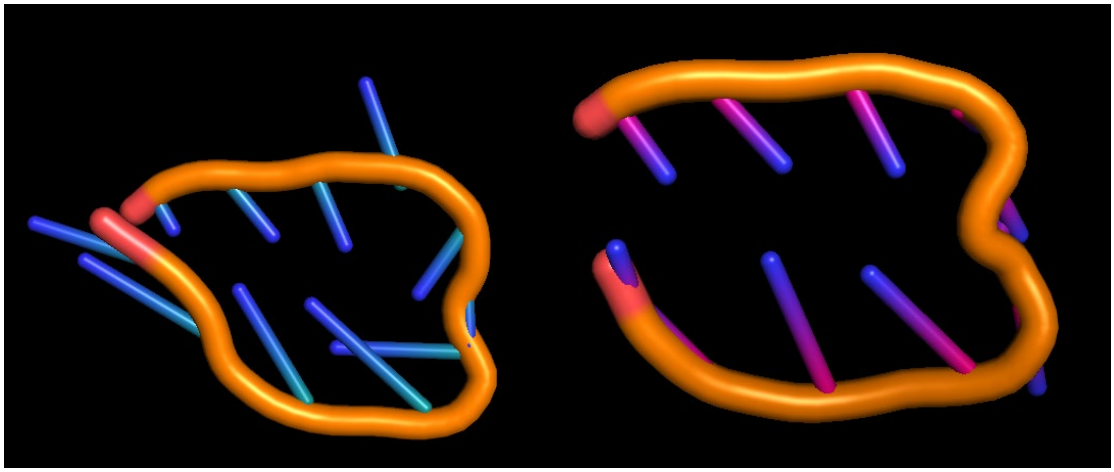


Figura 23. Muestra real (izquierda) y muestra artificial (derecha) del *run* 9

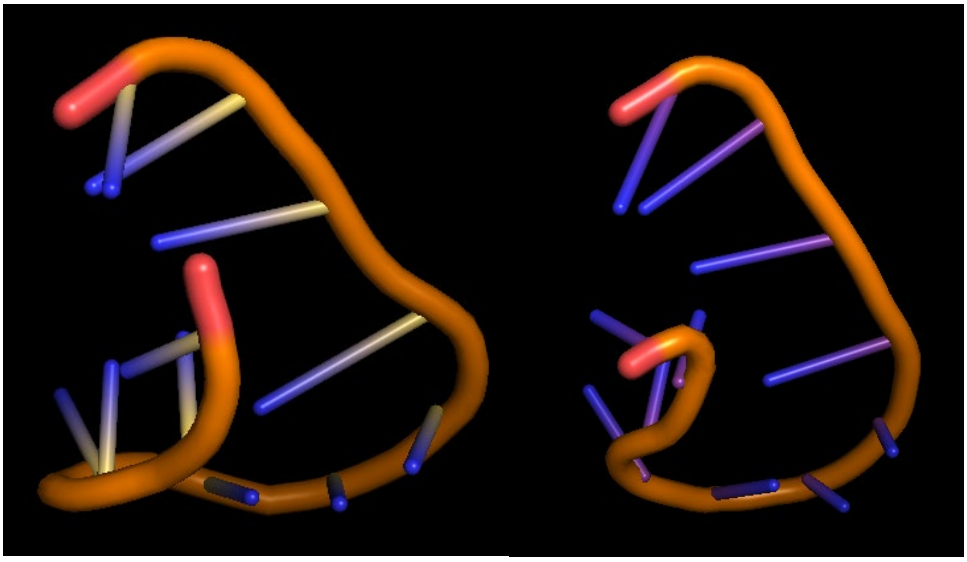


Figura 24. Muestra real (izquierda) y muestra artificial (derecha) del *run* 12

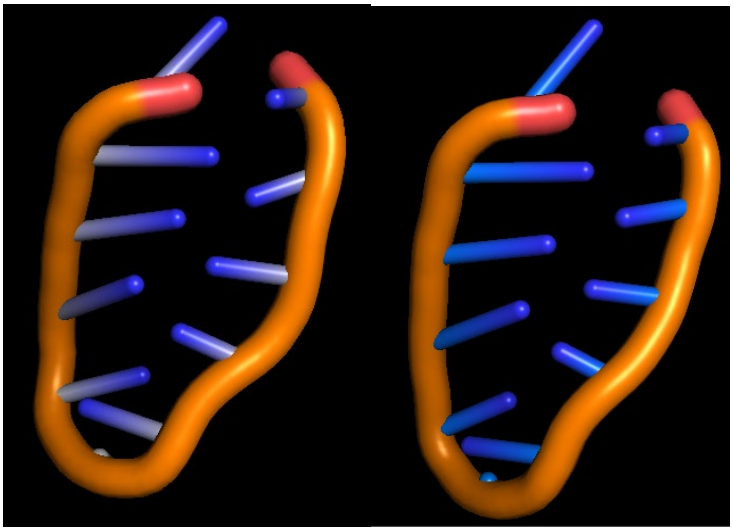


Figura 25. Muestra real (izquierda) y muestra artificial (derecha) del *run* 16

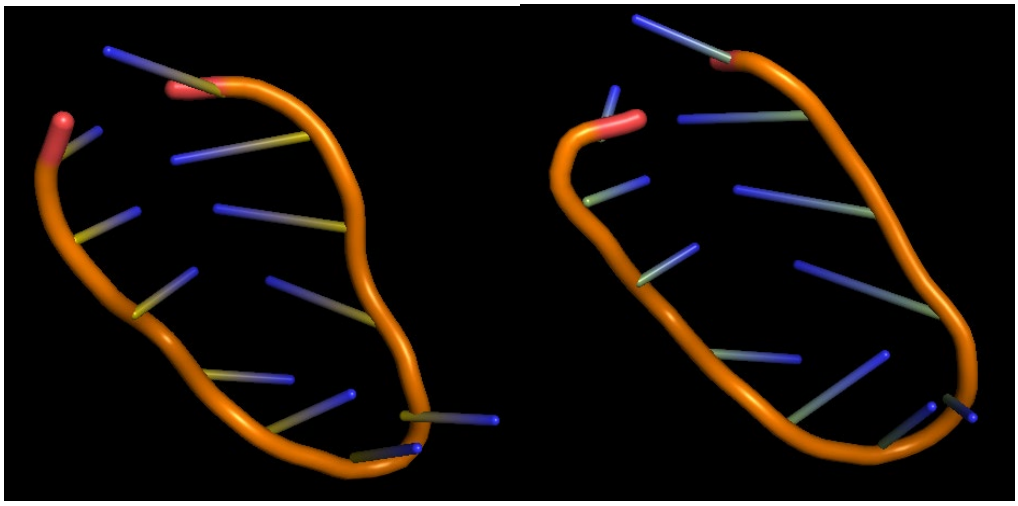


Figura 26. Muestra real (izquierda) y muestra artificial (derecha) del *run* 22